Support For Host Anycast, Priorities And Naming of Link-Local Addresses in IPv6

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Sameer Shah

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

August 1997

CERTIFICATE

This is to certify that the work contained in the thesis entitled Support For Host Anycast, Priorities And Naming of Link-Local Addresses In IPv6 by Shah Sameer M. has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dheeraj Sanghi

Dr. Dheeraj Sanghi,
Assistant Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology, Kanpur.

Abstract

Growth of the Internet raised issues that were not envisaged during design of its underlying protocol. Internet Protocol version 4 is faced with the twin problems of lack of scalability and flexibility. Efforts are ongoing in the design of next generation Internet Protocol (IPv6) that overcomes these drawbacks and provides future safe extensions. The base protocol and various supporting protocols have been standardized.

Additionally various issues are being worked on and of experimental nature. We have studied and implemented three such issues - support for host based anycast addressing, support for priority based traffic classes, and a name service for link local addresses. These require modifications at various layers of IPv6 stack - device, network, transport, BSD API, and application layers. The implementation can be used for experimenting with these issues to enable a better understanding of their feasibility and for tuning to individual requirements. These mechanisms were implemented and tested on a Linux 2.1.21 based IPv6 stack.

Acknowledgment

I would like to express sincere gratitude to Dr Dheeraj Sanghi, for providing me much needed encouragement and motivation throughout the duration of my stay. I thank him for providing me valuable insight and guidance in various topics, without which I would still be struggling with my limited vision. I am also indebted to him for allowing unrestricted use of his PC and office during the final semester. I thank Dr. Rajat Moona for helping me as a co-guide in the last semester and for enthusiastically teaching various systems concepts.

Thanks to Novell, Bangalore, for financially supporting me during the first 20 months of my course. I thank people on the various mailing lists ipng, netdev, linux-kernel, for the amount of knowledge they shared and the interesting discussions.

Two friends to whom I owe much are Nirav Shah and Biren Gandhi, it is only due to their friendship and support that I ever thought of making it to IIT. I cherish the long hours we worked together during our BE days. Jayaram greatly helped me during initial stages of the thesis and I learnt the basics of kernel hacking from him.

I thank the M Tech'95 and M Tech'96 batches for making my stay here a memorable one. I would like to particularly mention Samir Goel, Manu Thambi, K Srinivas and Deepjyoti Kakati for helping me in various ways. I also thank Vihari and Shyam for the nice company they provided.

My Parents, Bhai, Bhabhi and little Vidhi have always been very supportive and patient with me.

Contents

List of Figures

Chapter 1

Introduction

In this chapter we motivate the need for a next generation of Internet Protocol (IPv6). We then discuss the ongoing work towards the design and definition of IPv6. Finally an overview of the rest of the thesis is described.

1.1 Why IPng

Recent advances in networking, coupled with the availability of low-cost computing power have led to an explosion in size of the Internet. Additionally a wide spectrum of new applications, that depend on the support of widely varying service requirements, are now available. The present Internet Protocol [Pos81a] is unable to cope with the challenges posed by these developments. This led to the requirement for a next generation of IP, which fulfills present demands and provides future extensibility.

A basic problem with the present IP (IPv4) is growth in the number of hosts and networks. When IPv4 was designed, the developers could not foresee the kind of growth witnessed today. Surveys towards the beginning of 1996 indicate that there are over 9 million hosts in 2,40,000 domains, spread over nearly 1,00,000 networks. In the early 90s, Internet has been doubling its size every 12 months. This has resulted in problems with addressing and routing.

Theoretically, a 32-bit IPv4 address can address over 4 billion hosts spread over 16.7 million networks, but actual address assignment efficiency is far less [Hui94]. The problem is compounded by classification of IP addresses into three classes - A, B and C. Assignment based on an inflexible class structure results in wastage of large number of addresses i.e., which are assigned but not used. Projections in 1990 indicated that if assignment continued at the then rate, the class B space would be exhausted by March 1994 [BM95]. Hence further assignment from class B addresses was restricted, and the solution adopted was to assign multiple class C addresses to organizations of moderate size. Revised estimates in 1994 put the exhaustion of IPv4 space between 2005 and 2011.

IPv4 supports a primitive two level hierarchy with each address comprising a host id and network id. The route to each IP network should be available in routing tables of backbone routers. As there was no aggregation beyond the level of a network, each network occupied a separate routing table entry. With increased assignments from the class C address space, number of class C networks grew rapidly. This led to explosion in the size of routing tables of the critical backbone routers, and associated processing overheads during forwarding of packets and updation of tables.

One solution to the routing problem was suggested in [FLYV93]. It proposed use of Classless Inter Domain Routing (CIDR) which eliminates concept of classes and treats the 32-bit address as a continuous bit-maskable number. It allows the aggregation of multiple network ids in a single routing table entry. CIDR has largely influenced the IPv6 addressing architecture.

In addition to problems with addressing and routing, IPv4 lacks the basic mechanisms to support the requirements of a variety of new applications. Future growth in the Internet would be driven by newer markets, such as nomadic computing, networked entertainment and network controlled smart devices. A primary requirement of many such applications is support for Qualities of Service (QoS), where applications specify a desired QoS in terms of bandwidth, end-to-end delay bound and tolerable loss, and the network either guarantees this or refuses the connection. The protocol processing overhead should be minimized. It should support mobility of one or both nodes in a conversation. It should provide mechanisms to ease the configuration of addresses and other network layer information. Security mechanisms are a basic requirement for the growth of Internet commerce.

IPv4 Address Extension

Some solutions have been suggested to extend the life of IPv4, so as to delay transition to IPng. One proposal suggested careful renumbering of major portions of Internet to recover and minimize unused addresses. But the effort involved would be enormous [Gro94].

Another proposal was to use a dual network addressing scheme [WC92] using internal addresses that are unique within a network, and a limited number of global IP addresses that are shared among internal nodes. The latter are dynamically assigned when conversation with external nodes is desired. Problems with this approach are restrictions on the number of simultaneous external connections, and lack of globally unique addresses for internal nodes.

Another proposal was to use the remaining class A addresses conservatively and subnet them further for use with CIDR. Additionally, reserved addresses could be used to extend the length of an IP address to more than 32 bits.

None of these techniques offer a permanent solution to the address space crunch and only postpone exhaustion by few years. Additionally they do not address requirements of emerging applications such as QoS, mobility, autoconfiguration, security etc.

1.2 IPv6 Development

IPv6 [DH95] is a result of the efforts of several IETF working groups and proposals in the last few years. A white paper was issued in [BM93] to solicit the perceived requirements of an IPng, from various viewpoints. Based on the received feedback, several criteria were outlined in [PK94] - support for atleast 10^9 networks and 10^{12} hosts, architectural simplicity, free availability of specifications, topological flexibility, performance at par with commercial high-speed media, robustness, flexible transition, media independence, underlying service based on datagrams, ease of configuration, security, multicast support, extensibility, support for service classes, mobility, etc.

IPv6 is designed as an evolutionary step from IPv4. It can be installed as a normal software upgrade in internet nodes. It interoperates with IPv4 as long as the IPv4 address space does not exhaust. It is designed to run well on the spectrum of available link technologies, from low bandwidth networks such as wireless to high performance networks like ATM.

Proposed standards for the base protocol and various supporting mechanisms are available [DH95, CD95, NNS96, TH95, TN96, GN96, GTBS97, Atk95c]. But there are various issues that are still being worked out in the IPNG working group and of an experimental nature. These include end system designator (EUI) based addressing, mobility, host anycasting, qualities of service, naming of link local addresses, dynamic host configuration, operation over non broadcast multiaccess links, multihoming, dynamic readdressing, support for site-local addresses, router renumbering etc.

In this thesis, we have worked on three of these issues. These are support for host anycast addressing, support for classes of traffic based on priorities, and a name service for link-local addresses. The implementation can be used for experimenting with these issues so as to enable a better understanding of their feasibility and for tuning to individual requirements.

We have used the Linux 2.1.21 OS for implementation and testing of these mechanisms due to several reasons - Linux is a freely distributed, POSIX compliant OS, available on various hardware platforms. The base IPv6 protocol and supporting mechanisms have already been implemented in Linux and a set of applications and network utilities have been ported to work over IPv6. Source code for the kernel, related documentation [Joh96] and supporting utilities are freely available. We also benefited from past experience in implementing the basic IPv6 protocol and tunneling mechanisms [Jay96].

1.3 Organization of Thesis

In **Chapter 2** we discuss the salient features of IPv6. An outline of the base protocol is given which includes the IPv6 header, addressing architecture, routing, QoS support. We briefly discuss other support mechanisms - transition, neighbour discovery protocol, security mechanisms and path MTU discovery protocol.

In **Chapter 3** we discuss our implementation of host anycast support in IPv6. An anycast address identifies a set of interfaces with the property that a packet sent to this address is delivered to only one of them. Two problems in using anycast addresses are that they cannot be used as source address in any case, and they cannot be used as destination address if the protocol requires maintenance of state (like TCP). Even if a protocol does not require state (like UDP), applications may insist that all packets reach the same destination. In this chapter, we discuss a solution to this problem. The requirements at a host to support anycasting are specified. Actions are specified at the network, transport and API layers. The semantics of anycast usage primarily depend on the nature of transport layer. We have focussed on mechanisms in TCP [Pos81b] and UDP [Pos80].

Chapter 4 discusses implementation of priority support in IPv6. IPv6 provides a 4-bit priority field, but the definition and intended usage are still experimental. In a decentralized Internet, different providers/routing domains can implement priority differently based on the available resources, perceived application behaviour and local policy. There being no single approach acceptable to all, IPv6 priority is loosely defined. We discuss various issues related to IPv6 priority support. Based on this an extended definition is proposed and various priority mechanisms are suggested.

Chapter 5 describes a naming service for IPv6 link-local addresses. Link-local addresses are designed for use on a single link for purposes of neighbour discovery, automatic address configuration etc. These can be configured without a need for any formal setup. A problem with link-local addresses is of associating corresponding system names. This association cannot be stored in DNS, without greatly increasing its complexity. Manual configuration is undesirable. We describe a solution to this problem. A service is implemented on top of UDP that uses a multicast group of servers running on individual nodes. Necessary implementation details are described and recommendations for future implementations are provided.

Finally, in **Chapter 6** we conclude by summarizing the results of our efforts and discuss areas of future work.

Chapter 2

Salient Features of IPv6

We study how IPv6 and its supporting mechanisms attempt to fulfill the wide set of criteria mentioned earlier. The major improvements and new features are discussed.

2.1 Base Protocol

2.1.1 IPv6 Header

An IPv4 datagram contains a network layer header followed by upper layer data. IPv6 provides multiple network layer headers [DH95]. The base header is present in all datagrams and occurs before all other headers and upper layer data. Between base header and upper layer data, there may be several extension headers, that are optional. The base header is simplified compared to the IPv4 header. Some fields have been dropped or made optional to reduce the per hop processing cost and limit the bandwidth cost of IPv6 header. Though the IPv6 address size (128-bit) is four times that of IPv4, size of the base header is only twice the size. Fields are aligned to facilitate extraction of maximum performance for 64-bit architectures.

Various extension headers have been defined, each identified by a distinct value. This value is encoded in the Next Header field of a preceding header. The base header and all extension headers carry a Next Header field, which form a chain of headers terminated at the upper layer data. Currently defined headers are - Hop-by-Hop Options, Routing (Type 0), Fragment, Destination Options, Authentication and Encapsulating Security Payload. None of these headers, except the Hop-by-Hop Options header, is examined by any intermediate router. Thus information intended for end hosts does not incur processing in routers.

Header options have been encoded so as to allow efficient forwarding, less stringent limits on option length and greater flexibility to introduce new options in future. Each option carries a 3-bit field that indicates a default action when a node does not recognize the option; and the nature of the option whether it can change enroute. The latter information is used by security mechanisms.

IPv6 also provides a mechanism to send jumbograms, packets of size greater than 64k octets. Note that the IPv6 base header provides only a 16 bit Payload Length field.

2.1.2 Addressing Architecture

Size of an IPv6 address is 128 bit. This supports more levels of addressing hierarchy and much greater number of addressable nodes. It simplifies address autoconfiguration. Based on a study conducted by Huitema [Hui94] regarding efficiency of current addressing architectures, he concludes that 128 bits can accommodate between $8x10^{17}$ to $2x10^{33}$ nodes. The IPv6 addressing architecture is defined in [HD97].

An IPv6 address is an identifier for an interface or set of interfaces. An interface can be assigned multiple IPv6 addresses. Additionally a single address can be assigned to multiple physical interfaces *if* the multiple physical interfaces are presented to IP layer as a single interface. Though a subnet is associated to one link as in IPv4, more than one subnets can be assigned to the same link.

There are various ways to represent IPv6 addresses in text strings. A preferred form is 'x:x:x:x:x:x:x:x', where the x's are hexadecimal representation of the eight 16-bit pieces of the address. Leading zeros within a piece need not be shown. In another form, multiple groups of 16-bit pieces of zeros can be replaced by a '::' notation. This can appear only once in a string. For convenience in handling of IPv4 based IPv6 addresses, another form is 'x:x:x:x:x:x:d.d.d.d', where the x's are hexadecimal representation of the six high-order 16-bit pieces of the address and the d's are decimal values of the four low-order 8-bit pieces of the address. A 'prefix' is any non zero number of contiguous bits starting from the left-most bit. Prefixes can be represented as 'ipv6-address/prefix-length' for example FE80::C00F:8F/64.

The variable-length field comprising leading bits of an address that indicates the type of an address is referred as the Format Prefix (FP). Present assignment of format prefixes is available in [HD97]. Only 15 % of the address space is initially allocated, while the rest is reserved for future use.

Basically there are three types of IPv6 addresses - unicast, anycast and multicast. These are described below.

Unicast addresses

A unicast address identifies a *single* interface. Various forms of unicast addresses are defined - aggregatable global unicast address, NSAP address, IPX hierarchical address, site-local address, link-local address, and IPv4-capable host address. We discuss the main categories of addresses here.

Aggregatable global unicast addresses are defined in [HOD97]. This supports both the current provider based aggregation and a new type of aggregation called *exchanges*. An exchange is a special kind of provider that connects to multiple long-haul providers and independently assigns addresses to subscribers. Aggregatable global unicast addresses will provide efficient routing aggregation for sites that directly connect to providers or that connect to exchanges. The address format is shown in Figure 1.

```
| 3 |  13 |    32     |   16   |            64 bits             |
+---+-----+-----------+--------+--------------------------------+
|001| TLA |   NLA*    |  SLA*  |         Interface ID           |
+---+-----+-----------+--------+--------------------------------+
```

Figure 1: Aggregatable Global Unicast Address

Here 001 is the FP, TLA identifies a Top Level Aggregator, NLA* identifies Next Level Aggregator(s), SLA* identifies Site-Local Aggregator(s), and the Interface ID identifies a particular interface on a link. Delegation of prefixes within NLA and SLA space works similar to IPv4 CIDR delegation [FLYV93]. Interface ID is required to be of 64 bits and is constructed in IEEE EUI-64 format [IEE97]. Interface IDs may have global scope or local scope.

Two types of *local-use* unicast addresses are defined - Link-Local, used on a single link, and Site-Local, used in a single site. The former is identified by a prefix FE80::/64, while the latter is identified by a prefix FEC0::/64. Link-local addresses can be used for for purposes of automatic address configuration, neighbour discovery or if when routers are present. Site-Local addresses can be used for addressing inside a site without the need to obtain globally unique prefix.

IPv4-capable IPv6 addresses are useful during transition from IPv4 to IPv6. An address of the type '::d.d.d.d' (or equivalently '::x.x') is termed an IPv4-compatible IPv6 address and is assigned to hosts and routers that dynamically tunnel IPv6 packets over IPv4 routing infrastructure. An address of the type ::FFFF:d.d.d.d is termed IPv4-mapped IPv6 address and is used to represent addresses of *IPv4-only* nodes as IPv6 addresses.

Anycast addresses

An anycast address is an address assigned to more than one interface, with the property that a packet sent to this address is routed to the nearest interface having the address.

Multicast addresses

A multicast address identifies a group of nodes and a packet sent to this address is delivered to all the nodes. All multicast addresses have a format prefix of 0xFF. Each multicast address contains - 4-bit scope, 4-bit flags and 112-bit group ID. Scope field limits the scope of a multicast address in a limited region such as node-local, link-local, site-local, organization local and global. The flags field classifies multicast addresses into - permanently (well-known) and non-permanently (transient) assigned addresses. The meaning of a permanent multicast address is independent of its scope and it can be assigned to popular services such as all NTP servers. Transient addresses are meaningful only within the specific scope.

2.1.3 Routing

The routing function is identical to IPv4 routing under CIDR, except for the use of 128-bit addresses. Current routing algorithms such as BGP, OSPF, IDRP etc. can be used for IPv6 with few changes.

Additionally IPv6 supports simple routing extensions that provide powerful functionalities such as provider selection (based on policy, performance, cost etc), host mobility etc. A separate extension header, the routing header, can be used to list one or more intermediate nodes to be visited on a route to the destination. Routing header allows defining multiple types of source routing. Type 0 source routing defined in [DH95], allows specification of atmost 24 intermediate hops and a 24-bit bitmap specifies type of each hop - whether strict or loose. The security inherent to source routing [Bel89] is avoided by use of the IPv6 authentication and/or security mechanisms [Atk95a, Atk95b].

2.1.4 QoS Support

IPv6 provides support for Qualities of Service traffic in two ways - a 4-bit priority and a 24-bit flow label. The priority is discussed in detail in Chapter 4. We discuss the flow label here.

Flow labels can be used to request special handling in routers such as non-default QoS or real-time service. Emerging applications such as video on demand, netphone, depend on the degree of consistent throughput, delay and/or jitter. Such applications can use flows that can be assigned fixed resources in the network (allocation may be statistical). A flow is a sequence of packets sent from a particular source to a particular destination that need the same type of service from routers. The exact service specification will be conveyed using signalling protocols such as RSVP [Wro96].

Flow labels can be used in other ways, for example routers may opportunistically set up state for a flow though no signalling protocol was used. They may cache the result of initial datagram processing so as to speed up processing of later packets of the same flow. It has also been suggested to use flow labels in a hop-by-hop manner for example to carry a tag used by tag switching protocols so as to speed up the forwarding function.

2.2 Support Mechanisms

2.2.1 Transition

Transition is a key issue for the successful deployment of IPv6. Two primary requirements are - flexibility of deployment, and ability of IPv4 nodes to communicate with IPv6 nodes. A controlled transition of all IPv4 nodes to IPv6 is not possible for the current scale of Internet. Applications designed to work on IPv4 should continue working without change.

IPv6 defines two basic mechanisms for transition [GN96].

Dual IP layer To provide complete support for both IPv4 and IPv6 in hosts and routers. Such nodes can communicate with IPv4 nodes using IPv4 packets and with IPv6 nodes using IPv6 packets. A dual node may be configured with both IPv4 and IPv6 addresses. If such a node also implements the tunneling function described next, then the two addresses resemble each other and the IPv6 address is termed as an IPv4-compatible IPv6 address.

IPv6 over IPv4 tunneling IPv6 packets can be encapsulated in IPv4 headers and tunneled over IPv4 routing infrastructures. While the IPv6 infrastructure is being deployed, existing IPv4 routing domains can remain functional and can carry IPv6 traffic using tunneling. The two end-points of a tunnel are responsible to encapsulate/ decapsulate IPv6 packets. Based on the method used to determine IPv4 address of remote tunnel endpoint, tunneling is classified into - automatic and configured tunneling. In automatic tunneling, the Destination node has a IPv4 compatible IPv6 address and so it acts as the tunnel end-point. Configured tunneling is required when the Destination node does not support tunneling, instead the tunnel end-point is an intermediate router. In this case IPv4 address of this router needs to be configured at the encapsulating node.

2.2.2 Neighbour Discovery

A neighbour discovery protocol is used by IPv6 nodes to discover various information about neighbours [NNS96]. This protocol combines various IPv4 protocols such as ARP, ICMP Router Discovery, ICMP redirect in a single mechanism and additionally provides other improvements. It defines mechanisms for - router discovery, prefix discovery, parameter discovery, address autoconfiguration, address resolution, next-hop determination, neighbour unreachability detection, duplicate address detection and redirect. The protocol defines five new ICMP message types for its working. Providing these mechanisms on top of IPv6 layer allows use of the IPv6 security mechanisms with the protocol.

2.2.3 Security

Two mechanisms are defined to meet the security requirement - Authentication Header (AH) and Encapsulating Security Payload (ESP) [Atk95c, Atk95a, Atk95b]. These options can be used in isolation or in combination, to provide different levels of security.

Authentication allows the receiver of a datagram to ascertain that the claimed sender is the actual sender and to ensure datagram integrity. AH allows conveying information related to authentication. The extension header is defined to be algorithm independent and can support various algorithms. But implementation of the keyed MD5 algorithm is proposed to ensure interoperability in the Internet.

ESP provides confidentiality and integrity to IP datagrams. This implies that only the intended recipient can know what is sent but unintended parties cannot determine the contents. This requires encryption of data using cryptographic keys. The use of DES CBC algorithm has been proposed for universal interoperability.

Though several proposals for key distribution algorithms exist, no specific mechanism is yet standardized.

2.2.4 Path MTU discovery

IPv6 does not permit intermediate nodes to fragment datagrams, and encourages hosts to learn the path MTU of destinations. This reduces the processing overhead in routers, additionally attempt is made to avoid fragmentation in most cases. Arguments against fragmentation are discussed in [KM87]. Fragmentation leads to inefficient resource usage in terms of processing cost and bandwidth for additional header information. As IP reassembly is not robust, loss of a fragment causes the entire upper layer packet to be retransmitted. Also once fragmented, reassembly occurs only at the destination, thus benefit of higher MTU links after a low MTU link are not obtained.

A Path MTU discovery algorithm for IPv6 is proposed in [MDM96]. A source starts with an initial MTU equal to the MTU of outgoing link. If the packet is too large for an intermediate router, it is dropped and an ICMP packet too big message is sent to source. This also conveys the actual MTU of its outgoing link. The source uses this value in future packets. If this fails again, the cycle repeats itself until the exact MTU is discovered. The mechanism also allows a host to learn increases in MTU in the face of routing topology changes.

Chapter 3

Implementation of Host Anycast Support

3.1 Introduction

The IPv6 addressing architecture [HD97] defines a new type of address - the *anycast address*. This address is similar to a multicast address in the sense that it is assigned to a group of interfaces (typically belonging to different nodes). But it is similar to unicast address in that a packet sent to this address is delivered to only one interface. A packet sent to an anycast address is routed to the "nearest" interface having that address, according to the routing protocols' measure of distance.

Some uses of anycasting in router addressing are - to identify set of routers belonging to an organization providing Internet service, e.g., to enable provider selection; to identify set of routers attached to a subnet; or those providing entry into a particular routing domain, e.g., the home network for mobile nodes. In all these scenarios an anycast address is obtained from the network prefix assigned to the particular level of hierarchy the routers belong to and do not require special handling in the backbone routers.

There are various scenarios of host addressing where anycasting can be beneficial. An example is load sharing - all servers providing a service can be members of an anycast group, and a client's request is automatically delivered to the nearest server. Another example is fault tolerance - in case of failure of one server, requests are automatically routed to another node in the group. Use of anycasting leads to optimal routing paths and better network utilization.

Despite the benefits of a host anycasting service, [HD97] prohibits assignment of anycast address to IPv6 hosts till more experience is obtained and related problems are solved. There are two main problems -

- For every anycast address, there is a host-specific routing table entry in routers in the entire topological region covered by these hosts. This is acceptable if the region is small, like an Intranet, or a campus-wide network. But if there is a worldwide anycast group, then this implies having a host specific entry for an address in Internet backbone routers. This leads to routing table explosion and associated overheads in critical backbone routers.
- Successive packets sent to an anycast address may not all be delivered to the same node. There are two implications of this property. An anycast address cannot be used as the source address in an IPv6 packet due to inability to identify the originating node in case of error conditions. Stateful protocols cannot rely on the use of an anycast destination address throughout a session.

  Additionally a packet can be received by multiple anycast nodes due to duplication and misrouting. So more than one node can reply simultaneously.

Amongst the two problems, the former is unavoidable given the addressing architecture of IPv6. We believe that the use of anycast addresses will be restricted to organizational Intranets, and through controlled route leaking amongst Internet Service Providers. There may also be a mechanism to create global anycast groups in a regulated fashion. In this paper we provide a solution to the latter problem.

A host anycasting service was proposed for IPv4 networks by Partridge, et. al. [PMM93]. They discuss ideas on semantics of usage, and give suggestions for implementation. They allow use of an anycast source address in IP datagrams. A separate class of IP address is proposed for anycast allocation. This way TCP can *recognize* an anycast address, and treat this differently. An anycast destination address can only be used in the first segment. The remote host uses a non-anycast source address in the reply. The initiating host treats anycast address as wildcard (as they are recognizable), so the reply matches the request. The requesting node then replaces the anycast destination with the returned source address in local state. Future exchanges use only the non-anycast address. The drawback of this scheme is use of a separate space for anycast allocation. This results in huge routing overheads, as they cannot be aggregated even in case of router addressing. The scheme has not been implemented in IPv4 networks.

Bound and Roque [BR96] propose a solution for a IPv6 based host anycasting service. A new Destination option is proposed - the Source Identification option. A node which responds to a request addressed to an anycast address places its unicast address as IP Source address, and also indicates its anycast address using the new option. TCP and stateful applications on top of UDP, can use the value of this option for demultiplexing and learning the unicast address of the peer. Further communication solely uses unicast addresses. The authors have outlined actions at a host receiving the Source-Id option.

In this chapter, we extend the anycast model of Bound and Roque in several ways. We define the complete implementation at both ends of a communication that uses host anycast addresses. A set of requirements for a functional and flexible host anycasting service are provided. Such a specification is necessary in order to avoid ambiguity in interpretation and to define a correct implementation. In case of TCP we allow use of the proposed option to identify the anycast address of a host doing an active connect. When a Source-Id option is used, it *must* be used in demultiplexing. The earlier model [BR96] suggests using this as an optional mechanism. Changes to the BSD API are provided to allow use of anycasting in a flexible manner. Using the proposed mechanism, many existing TCP and UDP applications continue to work unchanged. Applications that maintain state on top of UDP, across multiple TCP connections, or use end-point addresses at the application layer will need to be modified.

Our implementation allows merging of Source-Id option with the Binding Update option proposed for mobility support in IPv6 [JP96]. Binding Update conveys information about the care-of addresses associated with a mobile node's home address. An additional bit can be used to indicate the anycast case.

These requirements were implemented and tested in a LAN environment among a set of machines. Normal usage cases of applications like telnet, ftp, rlogin continue to work unchanged (some of these applications depend on network addresses in special cases which may need to be modified).

The rest of the chapter is organized as below. In Section 3.2, we give a brief background relevant to rest of the chapter. This includes description of TCP and UDP, and how demultiplexing works. Section 3.3 describes the Source-Id option, an outline of the proposed mechanisms for TCP and UDP, and discusses the issue of security. Section 3.4 discusses modifications required at various layers in the IPv6 stack.

3.2 Background

A brief description on the working of TCP, relevant to the rest of the chapter is given. Implementation of datagram demultiplexing is also discussed.

3.2.1 TCP

TCP is an end-to-end, reliable, connection-oriented transport layer, that provides resequencing and flow control [Pos81b]. It maintains state at the end-points of a

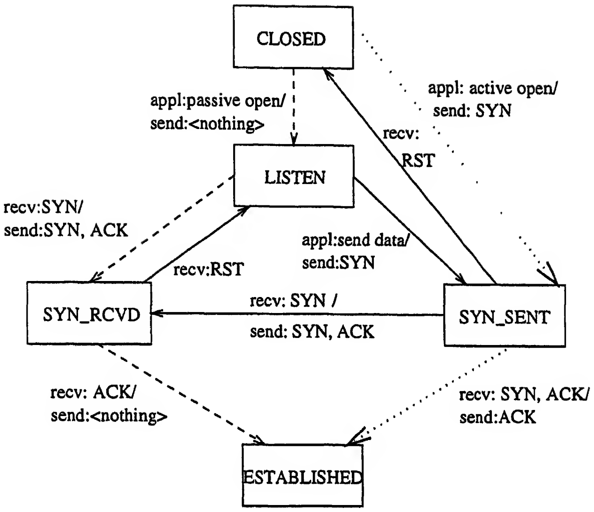

Figure 2: TCP State Processing during Initial Handshake

connection and provides mechanisms for connection establishment, data exchange and connection release/abort. The TCP state transition diagram is described in [Pos81b]. A modified subset [Bra89] showing transitions during the initial three way handshake is shown in Figure 2. Here dotted lines indicate normal transitions for clients and dashed lines are normal transitions for servers.

A server opens a connection in passive mode, enters LISTEN state and waits on connection requests from clients. A client doing an active open sends a segment with the SYN bit set, advertising its initial send sequence number (ISS); and enters SYN_SENT. The response to a SYN is a segment with both SYN and ACK bits set, acknowledging the clients' ISS and advertising the server's ISS. The server enters SYN_RECEIVED state. On receiving a response, the initiating client verifies the acknowledged sequence number, acknowledges the server's ISS in a reply segment with ACK bit set, and enters ESTABLISHED state. On receiving a valid ACK from the client, the server moves into ESTABLISHED state from SYN_RECEIVED.

TCP provides a mechanism (using RST bit) to reset the peers' connection when a packet is received for which no corresponding state exists locally. This provides recovery in case of loss of synchronization. TCP also allows the case of a simultaneous active open at both ends.

3.2.2 UDP

UDP provides a simple datagram oriented transport service - the only services it provides over IP are checksumming of data and demultiplexing by port numbers [Pos80]. The mechanisms at the UDP layer are simple compared to that in TCP, and it is the responsibility of applications to maintain any required state.

3.2.3 Socket Demultiplexing

A socket refers to an end-point of communication in the Berkeley networking system [Ste90]. An association is a 5-tuple that uniquely identifies a connection comprising {*protocol, local-addr, local-port, foreign-addr, foreign-port*}. We refer to the *protocol* implicitly and refer to this as the four tuple in rest of this chapter. A protocol control block (PCB) is a conceptual data structure containing protocol specific information for a socket. There is usually a one-to-one correspondence between socket and protocol control block in networking implementations.

The local port and local address are set with the `bind()` system call. Most implementations do not allow more than one socket to bind on the same local port. But this can be done by specifying the `SO_REUSEADDR` socket option with `setsockopt()`. The remote port and remote address members in a four tuple can be set using `connect()` system call. When a socket is unconnected, the remote address and remote port are wildcards. A `connect()` on a UDP socket does not involve data exchange, but merely saves the remote address and remote port values in the PCB. A UDP socket can issue `connect()` more than once. Most implementations, including ours, implicitly convert the remote address to the loopback address if `connect()` is invoked with a wildcard remote address. Hence a connected socket cannot have a wildcard remote address.

Figure 3 shows possibilities for values in the four tuple [Ste94]. A $\surd$ indicates a specific value while, "*" indicates wildcard.

Loc-IP	Loc-port	Rem-IP	Rem-port	Description
$\surd$	$\surd$	$\surd$	$\surd$	restricted to one client
$\surd$	$\surd$	*	*	restricted to dgrams arriving on Loc-IP
*	$\surd$	*	*	all dgrams sent to Loc-port

Figure 3: Specifying the four tuple

The order of the three rows in the table is the order used in demultiplexing a received datagram to the corresponding socket. The first row with most specific binding is tried first, and the last row with least specific binding is tried last.

3.3 Proposed Solution

In this section we describe the proposed Source-Id option, and how TCP and UDP should use the option. We also discuss the issue of security.

3.3.1 Source Identification Option

The Source Identification option is used by a host to inform a peer about its anycast address. The client side uses an anycast address as the destination address, but the server cannot use anycast address as the source address. If it were to use only its unicast address, the client would not be able to demultiplex it properly. Hence, the server sends its anycast address in the Source-Id option, in addition to using its unicast address as the source address. The client can demultiplex using the received anycast address, and also notes down the unicast address for further communication. The proposed option is encoded in the Destination Extension Header of an IPv6 datagram. The option is shown in Figure 4.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
                                +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                |  Option Type  |  Option Len   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                         Identifier                            |
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4: Source Identification Option

Option Type

Identifies type of the option. Its value should be assigned by the Internet Address Numbering Agency (IANA). The most significant 3 bits are 110. So a node not recognizing the option type must discard the packet and, only if the packet's Destination Address was not a multicast address, return an ICMP Parameter Problem, Code 2, message to the packet's Source Address. The option data does not change en-route.

Bound and Roque [BR96] suggest a value of 000 for the most significant bits, which allows ignoring the option and further processing of datagram. But allowing further processing, when the option is not recognized will not be useful in most cases of TCP usage and will work in UDP only in some cases. It also leads to wrong demultiplexing in few cases.

Option Length

Length of the option data field of this option in octets. This will carry a value of 16.

Identifier

The anycast address of the node originating the datagram whose unicast address is indicated in the Source Address field of the IPv6 base header.

3.3.2 Requirements for TCP

The mechanism should ensure proper synchronization with use of anycast addresses. A common scenario consists of a client doing active open on the server's anycast address. Client then waits in the SYN_SENT state, for a matching response from server. The request is received by a server waiting either on the anycast address or on the wildcard address. In either case, the server response (containing SYN+ACK or an RST) *must* carry a Source-Id option to associate its unicast Source Address with its anycast address. At the receiving end, TCP must use the identifier in the option while demultiplexing. If a matching socket is not found, an RST is sent to the remote host's unicast address. In this example, the server response matches the client socket in SYN_SENT. The client then switches to using the server's unicast address for all further exchanges.

We also allow the possibility of using the anycast indication in an active connect. In this case, the socket already selects a unicast source address for use during the connection, but it also sends a Source-Id option with the first SYN segment to identify itself with the anycast group. A matching socket at the remote TCP will be in one of SYN_SENT or LISTEN states. If the state is SYN_SENT, this corresponds to a simultaneous open scenario. If the socket is in LISTEN, it moves to SYN_RECEIVED upon receiving the request. The anycast address received in the option is saved in the PCB (separate from the sockets' four tuple), and can be provided to application when requested.

The above discussion leads to following constraints on the interaction of TCP with host anycast addressing:

- An anycast destination can only be used in an active connect to a peer. Thus a segment with anycast destination is valid only if the SYN is set and the ACK or RST bits are not set.
- A Source-Id option can be sent either in an active connect request, or in response to an active connect request that was received on the anycast address. Thus a segment with Source-Id option is valid only if the SYN or RST is set. The RST was sent in response to an active connect to the anycast address.

The demultiplexing as mentioned above is conceptually simple. But there are few subtle cases resulting from TCP's reliability in the face of losses, old duplicates and retransmissions. We discuss the detailed algorithm later in this chapter.

3.3.3 Requirements for UDP

Applications should be able to specify the host's anycast address as a *source address* for outgoing datagrams. Since IPv6 does not allow anycast addresses as source addresses, the UDP entity must replace the anycast address by an unicast address before passing onto IP layer. It should also indicate that a Source-Id option be added, before actual transmission.

Upon receiving a datagram with Source-Id option, demultiplexing must use the identifier in the option as the remote host address. If no match is found, or the recipient application prohibits the option in some cases, the datagram should be discarded and an ICMP port unreachable message be sent to the sender on its unicast address. There is an exception to this case, that we discuss later. Bound and Roque [BR96] ignore the option if it is not selected by application and allow further processing. In this case demultiplexing can match another socket specifying a remote address equal to the unicast address of destination (provided other members of four tuple are equal). It is incorrect to allow this in a host implementing anycast support, as the intended recipient was a socket connected either to remote's anycast address or a wildcard.

An application may *notice* a change in the address of responding node and take necessary action, e.g., issuing a new `connect()` to remote's unicast address. The API should allow applications to know the anycast address received in a Source-Id option, on a per datagram basis.

This proposal requires modifications to existing *stateful* applications. We propose a further extension to minimize the amount of this change. A flag `state_req` can be provided to indicate whether application maintains state and wants UDP entity to guarantee communication with a unique node. The default value of this flag should be false. Following action is invoked when a datagram with Source-Id option is demultiplexed onto a *connected* socket and `state_req` flag is true:

- If this is the first datagram to be received on the socket, then replace the remote address in the four-tuple by its unicast address. Outgoing datagrams will then use only the unicast address

- The anycast server may continue sending datagrams with Source-Id option set. Using the demultiplexing proposed above, a match will not be found (usually), once the above conversion occurs. To overcome this problem, the original anycast address is remembered in the PCB (separate from its four-tuple) and demultiplexing algorithm is modified to allow this possibility.

- If it is not the first datagram to be received on the socket, it implies that previous datagrams used the claimed anycast address as a Source Address.

This occurs in case of improperly configured anycast hosts or an attempt to spoof the address. *Do not* change the remote address but log a system error. Nevertheless, datagram should be delivered to application.

3.3.4 Security Issues

A mechanism to allow dynamic change of a peer's address in transport end-points is subject to threats similar to the source routing attack. For better security the proposed mechanism should be used in conjunction with replay protection and Authentication Header [Atk95a] as suggested in the mobile case [JP96]. Note that the security association used in AH computation should correspond to the anycast group address, instead of the Source Address of the datagram.

We did not implement the interaction with security mechanisms because the IPv6 AH computation was not available in the OS. We note that some amount of weak security is inherent to the mechanisms we employ.

- TCP accepts a response to an active connect only if the acknowledged sequence number matches the ISS sent in the request. Spoofing an address requires an impersonator to guess the ISS used by sender. As sequence numbers are chosen with sufficient randomness [Bel96], it is difficult to guess the ISS, unless an attacking node lies on the same subnet as the sender. But in this case other forms of attacks are also available, and the threat is inherent to type of link technology rather than this protocol.

  Also note that this mechanism does not secure the case of a simultaneous open.

- For UDP a datagram with Source-Id option received on a connected socket with `state_req` set, causes a change in the remote address in the four-tuple only if it is the *first* datagram to be received.

Additionally there can be applications without concern for security issues, e.g., a simple time of day service; or some environments may have a level of trust. In such cases use of AH can be avoided.

When a datagram with Source-Id option is demultiplexed to a TCP socket, we use a flag to determine from one of three choices: AH not required, AH not required only if received in response to active connect (excluding the simultaneous open case), or AH required. For UDP the flag indicates one of two choices: AH required or AH not required.

3.4 Implementation

The implementation of host anycast support at the network, transport and API layers is discussed in this section.

Design Goals

The primary objective of the implementation is to provide a mechanism to support use of host anycast addressing in transport specific ways. The implementation should have minimal per-packet processing overhead. There should be minimal changes to protocol specifications.

3.4.1 Network Layer

The network layer encodes a Source-Id option in outgoing datagrams when indicated by the upper layer. On incoming datagrams it decodes the option, and passes the identifier to upper layer. Additional changes are discussed below.

Configuring an anycast address

We provide a mechanism to manually configure an anycast address for an interface. The scope of an anycast address can be one of : link local, site local or global. Duplicate address detection is not done when configuring an anycast address [TN96]. The address is added to a list of addresses for the interface. Packets received with a destination address equal to this address will be handed to upper layers.

Neighbour Discovery

Anycast addresses are treated just like unicast addresses for the purpose of Neighbour Discovery except for minor exceptions [NNS96]. Neighbour Advertisements sent in response to a Neighbour Solicitation are delayed by a random amount of time between 0 and MAX_ANYCAST_DELAY_TIME to reduce probability of network congestion. Also, Neighbour Advertisements carry a value of 0 for the Override flag. So, if multiple advertisements are received for the anycast address, the first one will be used, instead of the most recent.

Source Address Selection

The network layer prohibits selection of an anycast address as a source address in outgoing datagrams. When a request is received on anycast address, following are the steps to determine a unicast source address for use in the response.

- Find outgoing interface for the destination using Next Hop Determination algorithm [NNS96]. If no route found to destination, return error.
- Select an unicast address with appropriate scope from list of addresses assigned to the interface. If no such address available, select a unicast address

with appropriate scope from addresses assigned to other interfaces. If no such address found, return error.

- The outgoing interface may differ from that identified by the anycast address. In this case further communication with unicast address may use an entirely different route than the route taken for anycast address. This may not always be desirable, but is allowed.

3.4.2 TCP Layer

Actions at hosts sending or receiving Source-Id option are discussed below.

Background

A socket in LISTEN state maintains a queue of connections doing initial handshake. The remote address and remote port of socket in LISTEN state are *always* wildcards. Each connection doing handshake is represented as a variable of type `struct open_request`. When the final acknowledgment in the three way handshake is received, a PCB is allocated for the connection and the PCB is inserted in a system wide list of TCP PCBs. On receipt of a segment, this list is searched for a matching socket.

Sending Source Identification

A flag (sndany) is used to enable or disable Source-Id option on outgoing segments. If the flag is not set, anycast TCP packets are responded with RST packets. The Source-Id option is sent in two instances: First, request with anycast destination is received on a socket bound to the anycast or wildcard address. Second, a socket bound to anycast address initiates active open. In either case the local anycast address is remembered (as the variable `loc_anyaddr`) in the PCB, separate from the four-tuple. This is required in later demultiplexing, in special cases.

Receiving Source Identification

On receipt of a segment with Source-Id option, TCP must use the option value in demultiplexing. The exact demultiplexing is discussed later. For now assume a recipient socket is found in demultiplexing. The further processing depends on state of the socket. In all the following cases, a socket changes the remote address in four-tuple to the remote unicast address, and remembers the remote anycast address also (as the variable `rem_anyaddr`).

- If the socket is in SYN_SENT state and segment is a response to an active connect on an anycast address, the state is changed to ESTABLISHED.
- If the socket is in LISTEN state and segment is an active connect request from the anycast node, a new request is allocated on the listen queue and its state is set to SYN_RECEIVED.
- If the socket is in SYN_SENT state and segment is an active connect request, this corresponds to simultaneous active open. The state is changed to SYN_RECEIVED.
- Otherwise the option is ignored.

Examples

An example of active connect to remote anycast is shown in Figure 5. Conventions used in the figure are - Segment with (B,A,X) implies Source Address = B, Destination Address = A and Id in Option = X. For a socket in LISTEN state, its listen queue is shown alongside. S1: (A,B,X,Y) indicates that for socket 1, loc-unicast IP = A, rem-unicast IP = B, loc_anyaddr = X, and rem_anyaddr = Y. If loc_anyaddr is set, a Source-Id Option was sent to remote in the handshake. If rem_anyaddr is set, a Source-Id option was received from remote host in handshake.

	TCP A				TCP B, X
					S1:LISTEN(*,*)[]
a)	S1:(A,X, ,)	→	<SYN>(A,X,)	→	S1:LISTEN(*,*)[(B,A,X,)]
b)	S1:(A,B, ,X)	←	<SYN,ACK>(B,A,X)	←	S1:LISTEN(*,*)[(B,A,X,)]
c)	S1:(A,B, ,X)	→	<ACK>(A,B,)	→	S1:LISTEN(*,*)[] S2:(B,A,X,)

Figure 5: Active Connect on Anycast Destination

In line a) S1 on TCP A initiates active connect request to B's anycast address X and moves to SYN_SENT. TCP B matches this to S1 in LISTEN state, and responds with SYN + ACK and a Source-Id for X (line b). Upon receipt of this response, S1 on TCP A changes to ESTABLISHED state and replaces remote address by unicast address of B. The final ack is sent to unicast address B (line c). On receipt of this ack, a PCB is allocated for the request on listen queue and it changes to ESTABLISHED.

An example of active connect with local anycast address is shown in Figure 6

	TCP A, Y				TCP B
					S1:LISTEN(*,*)[]
a)	S1:(A,B,Y,)	→	<SYN>(A,B,Y)	→	S1:LISTEN(*,*)[(B,A, ,Y)]
b)	S1:(A,B,Y,)	←	<SYN,ACK>(B,A,)	←	S1:LISTEN(*,*)[(B,A, ,Y)]
c)	S1:(A,B,Y,)	→	<ACK>(A,B,)	→	S1:LISTEN(*,*)[] S2:(B,A, ,Y)

Figure 6: Active Connect with Local Anycast

Prelude to Demultiplexing

We discuss the modified demultiplexing algorithm in rest of this subsection.

A bind to anycast is treated similar to a wildcard bind due to following reasons.

- A socket listening on wildcard local address, prevents other sockets to bind on the same port for any of the unicast addresses. An anycast bind is similar to a wildcard bind, because it can select any of the host's unicast address in response to incoming request.

- For a connection doing initial handshake on a listen queue, all incoming segments are matched to four tuple of the socket in LISTEN state. A connection initiated with anycast destination receives future segments with unicast destination which should be matched to the anycast (or wildcard) listening socket.

  For the same reason, we do not allow a socket bound to a wildcard or anycast address to enter LISTEN state, if another socket bound to any of the local addresses (for the same local port), is in LISTEN state.

Following routines are often referred in further discussion -

- `get_sockany()` corresponds to demultiplexing described in subsection 3.2.3 and modified to consider local anycast address as wildcard. Input to this routine is a four tuple, but we show only two arguments - local address followed by remote address. It returns a socket matching the four tuple which is not in CLOSED state. An example usage is -

  `sk = get_sockany(seg->dst IP, seg->src-unicast IP);`

- `srch_listenq()` searches socket listen queue. Input consists of a socket and three tuple (local port is same as that of socket in LISTEN state) and returns a request doing initial handshake. We do not show local port in the discussion.

- `discard(seg)` - silently discard segment.

- `send_reset(seg)` - send reset to remote in response to segment, reset is sent only if RST is not already set on incoming segment.

- deliver(sk,seg) - deliver segment to socket.

Receiving a segment

Incoming segments are handled as follows -

```
tcp_rcv(seg)
{
    if (Source-Id option received)
        if (!(SYN set or RST set))
            discard(seg), return;

    if (seg->dst IP is anycast)
        tcp_dmuxany(seg);
    else
        if (src ident option received)
            tcp_dmuxdopt(seg);
        else
            tcp_dmux(seg);
}
```

Unicast Destination, No Source Identification

This is the simplest case in demultiplexing. It is shown below -

```
tcp_dmux(seg)
{
    sk = get_sockany(seg->dst IP, seg->src IP);
    if (not found)
        send_reset(seg), return;

    if (sk not bound to wildcard and anycast)
```

```
        deliver(sk,seg), return;

    if (sk bound to anycast)
1)      if (SYN set)
            send_reset(seg), return;
        else
            deliver(sk,seg), return;

    /* sk bound to wildcard local address */
    op_req = srch_listenq(sk, seg->dst IP, seg->src IP)
    if (op_req not found || SYN not set)
        deliver(sk,seg), return;
2)  if (op_req->loc_anyaddr is set)
        send_reset(seg), return;
    else
        deliver(sk,seg), return;
}
```

Due to the special way we handle anycast bind, a segment with unicast destination can match anycast socket. This occurs only if an active connect was previously received on the anycast address, so later segments with unicast destination cannot have SYN set (excluding retransmissions). This is checked in line 1).

An example of the anomaly handled in line 2) is shown in Figure 7.

Two active connection requests are initiated simultaneously from the same host, one on the remote's anycast address, and other on its unicast address. Both cannot be allowed to proceed, as it leads to the same four tuple. Uniqueness of the four tuple needs to be enforced at the anycast host. It allows only one to proceed - the one which was received earlier, and the other request is aborted with a reset. In the figure, the request on the anycast address was allowed to proceed, as it was received first.

	TCP A				TCP B, X
					S1:LISTEN(*,*)[]
a)	S1:(A,X, ,)	→	<SYN>(A,X,)	→	S1:LISTEN(*,*)[(B,A,X,)]
		...	<SYN,ACK>(B,A,X)	←	S1:LISTEN(*,*)[(B,A,X,)]
b)	S2:(A,B, ,)	→	<SYN>(A,B,)	→	S1:LISTEN(*,*)[(B,A,X,)]
	S2:Abort	←	<RST>(B,A,)	←	S1:LISTEN(*,*)[(B,A,X,)]
c)	S1:(A,B, ,X)	←	<SYN,ACK>(B,A,X)	...	
d)	S1:(A,B, ,X)	→	<ACK>(A,B,)	→	S1:LISTEN(*,*)[] S2:(B,A,X,)

Figure 7: Active Connect on Both Addresses of Anycast Host

Anycast Destination Address

The steps in demultiplexing a segment with anycast destination are shown below. Note that in this case, if RST is sent, the Source-Id Option is also sent.

```
tcp_dmuxany(seg)
{
    if (RST set || FIN set || ACK set || not SYN set)
        discard(seg), return;
    reply-unicast IP = select source address for response;
    sk = get_sockany(reply-unicast IP, seg->src-unicast IP);
    if (sk not bound to anycast and wildcard)
1)      if (sk->loc_anyaddr == seg->dst IP)
            deliver(sk,seg), return;
        else
            send_reset(seg), return;

    /* Is sending of src ident allowed */
    if (sk not found || sk->sndany not set)
        send_reset(buff), return;

    if (sk bound to anycast)
2)      if (bound anycast address != seg->dst IP)
```

```
            send_reset(seg), return;
        else
            deliver(sk,seg), return;

    /* sk bound to wildcard */
    op_req = srch_listenq(sk, reply-unicast IP, seg->src-unicast IP);
    if (op_req not found)
        deliver(sk,seg), return
3)  if (op_req->loc_anyaddr == seg->dst IP)
        deliver(sk,seg), return;
    else
        send_reset(seg), return;
}
```

A scenario where 1) is true is when both end-points of a connection, initiated through anycast address, are in ESTABLISHED state and a duplicate of the first SYN is received. A scenario where 1) is false is similar to that shown in Figure 7, except that in this case, the request with unicast destination was received first and was allowed to proceed. Line 2) checks the case when a host has multiple anycast addresses. The check in line 3) is similar to that in line 1) except that in this case the other connection is not in ESTABLISHED state, but is doing initial handshake on a LISTEN queue.

Received Source-Id option

```
tcp_dmuxdopt(seg)
{
    sk1 = get_sockany(seg->dst IP, seg->src-anycast IP);
    sk2 = get_sockany(seg->dst IP, seg->src-unicast IP);
    if (sk1 not found)
```

```
            if (sk2 found and (sk2->rem_anyaddr == seg->src-anycast IP))
                deliver(sk2,seg), return;
            else
                send_reset(seg), return;

        if (sk1 disallows Source-Id)
            send_reset(seg), return;
        if (sk1 state not LISTEN)
        {
            if (sk1 state not SYN_SENT)
                log system error, send_reset(seg), return;
            if (RST set || sk2 not found)
                deliver(sk1,seg), return;
1)          if (sk2 state not LISTEN)
                discard(seg), return;
            /* sk2 is in LISTEN state, check listen queue */
            op_req = srch_listenq(sk2, seg->dst IP, seg->src-unicast IP)
2)          if (op_req found)
                discard(seg), return;
            else
                deliver(sk1,seg), return;
        }
        /* RST with src ident can't match anycast or wildcard */
        if (RST set)
            discard(seg), return;
        /* SYN with unicast dest can't match anycast socket */
        if (sk1 bound to anycast)
            send_reset(seg), return;
3)      if (sk2 state not LISTEN)
            discard(seg), return;
        /* here sk1 == sk2 */
```

```
    op_req = srch_listenq(sk1, seg->dst IP, seg->src-unicast IP)
    if (op_req not found)
        deliver(sk1,seg), return;
4)  if (op_req->rem_anyaddr != seg->src-anycast IP)
        discard(seg), return;
    else
        deliver(sk1,seg), return;
}
```

The conditions in lines 1), 2), 3) and 4) are similar - a SYN with Source-Id is received and a matching socket is available, but there exists another connection with same four tuple, either in synchronized state (cases 1 and 3) or on a listen queue (cases 2 and 4). If the current segment is delivered, this leads to two connections using same four tuple. Such a condition is usually checked at the remote host (it allows only one connection to proceed, and resets other), but this can occur for example if remote crashed and lost state. In this case the incoming segment is silently discarded. One of the two sockets will ultimately be reset by the peer, when loss of synchronization is detected on a retransmit attempt. Then the other connection can proceed.

3.4.3 UDP Layer

Sending end

An application can specify an anycast address for outgoing datagrams in two different ways, by binding to an anycast address, and by specifying an anycast source address as ancillary data in a call to `sendmsg()`.

We provide a flag (`sndany`) that an application can use to enable/disable sending of Source-Id option. When a socket is bound to an anycast address, value of the flag is tested for each outgoing datagram. A Source-Id option identifying the anycast address is sent only if it is set. If the flag is not set, the effect of a bind to anycast address is to allow reception of incoming datagrams destined for the anycast address, but outgoing datagrams use a unicast address without the option. This handling is similar to that for multicast addresses - a multicast address cannot be used as source address in outgoing packets [HD97].

When a socket is bound to wildcard address, it can receive datagrams on all local addresses. A robust server may use the destination address on incoming requests as the source address for outgoing responses. Receiving destination address on incoming datagrams and specifying source address on outgoing datagrams is accomplished using ancillary data to `recvmsg()` and `sendmsg()` respectively [ST97]. When an anycast address is specified as ancillary data to `sendmsg()` and `sndany` flag is set, a Source-Id option is automatically added. If `sndany` is not set, the `sendmsg()` returns an error.

Receiving end

Demultiplexing upon receiving a datagram *without* Source-Id option remains unchanged. This uses the algorithm specified in subsection 3.2.3. The corresponding routine is `get_sock()`.

The modified demultiplexing is shown below. This routine is handed all incoming UDP datagrams.

```
udp_rcv(dgram)
{
    /* Check for Source-Id Option */
    if (Src-Id not received){
        sk = get_sock(dgram->dst IP, dgram->src IP);
        if (found)
            deliver(sk,dgram), return;
        else
            send-icmp(dgram), return;
    }

    /* Source-Id received */
    sk = get_sock(dgram->dst IP, dgram->src-unicast IP);
1)  if (found && sk->rem_anyaddr == dgram->src-anycast IP)
        deliver(sk,dgram), return;
    sk = get_sock(dgram->dst IP, dgram->src-anycast IP);
    if (not found || Src-Id not allowed by sk)
        send-icmp(dgram), return;
2)  if (sk is connected && state_req set)
        if (first datagram on sk){
            sk->rem IP = dgram->src-unicast IP;
            sk->rem_anyaddr = dgram->src-anycast IP;
        } else
            log system warning message;
    deliver(sk, dgram);
}
```

Algorithm for UDP Demultiplexing

If the condition in 2) is true, a datagram with Source-Id option matched a socket with remote's anycast address in the four tuple. If this is the first datagram to be received, the remote address is set to unicast address and the anycast address is remembered in rem_anyaddr. This is used in case 1) to receive datagrams with the Source-Id option on the same socket.

When application reissues connect() on a socket or the state_req flag is changed from 0 to 1, rem_anyaddr is set to the unspecified address (all zeros) and a flag to indicate receipt of any datagrams on the socket is reset.

Receiving ICMP error

When an ICMP error is received for a previously sent datagram, the original datagram in error is returned as part of the ICMP message [CD95]. A socket corresponding to the message is determined by using the Source and Destination address/port pairs in the datagram for demultiplexing. If the original datagram contains a Source-Id option, the identifier in the option must be used instead of the Source Address.

3.4.4 Raw Socket

Raw socket provides a raw interface to the IPv6 layer that can be used, for example, by ICMPv6. The other use is to allow reading and writing IPv6 datagrams containing a Next Header field that the kernel does not process, e.g., OSPF over IPv6 or RSVP. Any datagram received at the IPv6 layer and destined for the local host, is delivered to all raw sockets in addition to corresponding transport layer (if one exists), if the specified Next Header field matches. All fields in a received IPv6 header (other than the version number and Next Header fields) and all extension headers are also made available to the application [ST97]. Hence no change needs to be made for receiving a Source-Id option, but applications are responsible to interpret this appropriately and may need to be modified.

For sending a Source-Id option on outgoing datagrams, handling is similar to the case of UDP, i.e., by setting sndany and either binding to an anycast address or specifying an anycast source address through ancillary data to `sendmsg()`.

3.4.5 Applications Programming Interface

An `ioctl()` is provided to assign an anycast address to an interface. Input to this routine is an interface index and an IPv6 address.

An ioctl() is needed to determine if the input address is a local anycast address.

Three socket options are required to get/set values of various flags. These flags control per socket anycast handling. All these options are defined at the IPPROTO_IPV6 level:

- IPV6_SNDANY - sndany flag controls sending of Source-Id option on outgoing datagrams. Possible reasons to disable this - higher demultiplexing overhead for datagrams which carry the option, or working of some stateful applications, etc.
- IPV6_RCVANY - rcvany flag controls demultiplexing of datagrams with Source-Id option. Values for this option correspond to those described in subsection 3.3.4.
- IPV6_STATE_REQ - state_req flag is specific to UDP.

getpeername() returns the remote address member of the four tuple. The remote address may change in some cases of anycast usage. Applications may use getpeername() to *notice* a change in the peer's address, e.g., after a connect() succeeds in TCP.

Applications can also learn a peer's anycast address using ancillary data. UDP can return the anycast address received in the Source-Id option on a per datagram basis in a call to recvmsg(). TCP uses a call to getsockopt() to receive one or more ancillary data objects in the buffer, for all the optional information it wants to receive. One of the optional information is peer's anycast address that is remembered in the PCB (as rem_anyaddr). In order to receive this optional information, applications must call setsockopt() to turn on a corresponding flag.

Chapter 4

Priority Support

4.1 Introduction

A priority mechanism can be used to allocate network resources differentially among traffic classes. Requirement for QoS support is recognized in IPv6 base specification [DH95]. It provides two mechanisms for QoS support - flow labels and priority. Flow labels can be used to specify quantitative requirements on desired QoS such as bandwidth, delay, incurred loss for individual flows. This requires use of appropriate QoS control service such as those described in [Wro97, SPG97] in conjunction with a signalling protocol such as RSVP [Wro96] to convey the desired QoS to routers. In contrast, priority usually specifies qualitative QoS requirements for groups of flows. It can be used for best-effort traffic to avoid the overhead of reservations.

Example usage of priority are - to identify customers with premium service, to indicate enhancement layers of video, to indicate interactive traffic etc. Priority can be used in various ways and different routing domains/providers may implement priority differently based on local policies, available resources and perceived user/application behaviour.

In this chapter we discuss issues related to IPv6 priority use. We implement various methods of priority separation based on an extended definition of the priority field.

IPv4 [Pos81a] provides an 8 bit type-of-service for QoS support. This consists of 4-bit TOS and 3-bit precedence. TOS can be used to decide on the route taken by datagrams [Bak95, Alm92]. The host requirements and router requirements specifications [Bra89, Bak95] mandates setting of appropriate TOS in datagrams and provision of TOS field in routing tables but TOS routing is not widely used. Precedence roughly corresponds to IPv6 priority field. The original intent was to provide high availability of critical defense data. But the concept can be applied to other scenarios. Precedence support in IPv4 was not used until recent years.

Deering and Hinden [DH95] define an enumerated priority with values ranging from 0 to 15, to identify priority *relative to packets from the same source.* Priorities 0 to 7 are assigned to traffic which backs off in response to congestion e.g., TCP. Priorities 8 to 15 are for non congestion-controlled traffic e.g., real-time packets sent at constant rate. It suggests example priority values for popular applications, in the congestion-controlled range. In the non congestion-controlled range least priority should be used for packets, a sender is most willing to discard in congestion and highest priority for data it is least willing to drop. No relative ordering is implied between the two classes.

Above definition was superceded by a different proposal [Hin97]. This removed reference to *drop* based priority because it allows sending data that is ultimately dropped and wasting network resources. The proposal declares 4-bits as rewriteable by routers/ISP for private purpose. Priority bits are not significant to receivers. Following recommendations are provided - low order bit is an Interactive bit, that can be set by sender to mean delay is more important than throughput. Routers/ISPs should use other bits before touching this bit. Priority affects only queueing, and not routing as it is re-writable.

We assign meaning to additional two bits and implement various queueing algorithms. A weighted fair queueing (WFQ) algorithm [DKS89, CSZ92] is used to distribute link bandwidth among some classes. Within these classes, various mechanisms are provided to separate sub-classes. One of these includes implementation of random early detection (RED) congestion control [FJ93], with differential drop thresholds. These mechanisms were implemented and tested in a LAN environment between two machines.

Rest of the chapter is organized as follows. In Section 4.2, we discuss issues in IPv6 priority usage. Section 4.3 provides an overview of our definition and algorithms. Section 4.4 describes implementation details. Results of the implementation are presented in Section 4.5.

4.2 Discussion

Issues related to definition and intended use of the IPv6 priority field are discussed. These are based on discussions in the IPng working group [Lis97b].

A basic requirement is that, any priority mechanism should be usable for both unicast and multicast traffic. One issue is that a 4 bit priority may be insufficient in some cases and more bits are needed. Additional bits could be obtained in various ways such as overloading the version field, using special flow label values, using a shim header before the IPv6 header etc.

Another issue is whether priority is set by hosts or routers. Both of these are allowed, applications set priority to indicate desired QoS while routers can overwrite this to implement local policy. Some bits such as the interactive bit are important at both the end-hops and hence should not be usually overwritten. One suggestion was to use two priority fields, one set by sender, other set by routers that indicates actual priority being offered to a packet.

We note that end-to-end priority selection is not fixed per application but depends on the current transaction, e.g., a WWW client may be interactive traffic at one instance, and later switch to bulk transfer.

Currently an isolated bit indicates interactive traffic. An alternative is to use 4 bits as an enumerated field and allow 16 values. One value can be assigned to interactive traffic, allowing rest for use in provider specific ways.

Priority can be compared within a single source (flow) or it can be comparable across sources. The definition in [DH95] provides only the former. This can be used, for example, among layers of a video from the same source, where an explicit resource reservation for the application is available in intermediate routers. But it is unsuitable for best-effort traffic, as it would require *dynamic* caching of state for ongoing traffic, resulting in increased router complexity and overheads. Additionally a mechanism to provide some form of QoS across sources of best effort traffic, without the overhead of reservations, is desirable. Hence priority is also comparable across sources. We concern only with the latter use, though this may implicitly achieve the former in few cases.

As different routing domains may implement priority differently, ingress/egress routers will need a mechanism to *map* priority of transiting packets, from those in previous domain to those in next domain.

Priority should not provide incentive for misuse such as users setting high priority in all data. An appropriate definition can mitigate this for example if interactive bit is set, a reduced throughput may be provided. Another solution is to implement usage based charging - high priority packets cost more than low priority packets. In absence of this, a suitable policing mechanism is required. It can punish non complying users for example by discarding packets or lowering the priority. Policing does not necessarily imply separate state for each source. It could be enabled in times of persistent congestion e.g., RED provides an efficient method to detect misbehaving users. Additionally having a rewritable priority is necessary for the same reason.

Implementation

Consider the likely ways in which priority can be implemented. One or more of these mechanisms can be used in isolation or in combination. Priority ordered queueing can be provided, where high priority packets are served ahead of low priorities or have high probability of timely arrival. This corresponds to either strict ordering, with starvation of low priorities; or non-strict ordering without starvation. Another method is to provide a drop based priority. In times of congestion, low priority packets are dropped or have higher drop probability. Priority can be used as a link sharing mechanism to distribute bandwidth among classes. We provide all these methods, the one being used can be configured by administration.

There are other implementations of priority. It can be used to map to appropriate priorities provided by some link layers. It can be used similar to IPv4 TOS. Though the current proposal does not recommend this use, there is no separate mechanism for TOS routing in IPv6 and some providers may need this mechanism.

Usage

Our implementation provides various ways of using priority. This is motivated from the wide applicability of a priority mechanism. Below we consider the various uses.

Cocchi et al. [CESZ91] argue for providing users a price vs performance tradeoff. Performance penalty to users of low priority is offset by reduced cost of service. Monetary penalty for users of high priority is offset by improved performance. They demonstrate that, it is possible to set prices so each user is more satisfied with the combined cost and performance of the network. Note some backbone providers in current now provide premium services based on IPv4 precedence.

Use of priority to indicate interactive traffic is a present requirement especially on low bandwidth links such as wireless. Target applications are X traffic, RPC, keystrokes etc. It can be used to provide low delay and also high availability for defense data, network management packets and routing protocol exchanges.

As a link sharing mechanism, priority can be used in various ways. It can identify subscribers provided with guaranteed capacity. It can isolate classes that use different congestion control algorithms such as those using TCP and those on top of UDP. It can be used to experiment with newer service models in the Internet [FJ95]. A new link sharing class can be created for the model, in restricted parts of Internet, and the service expanded, if it becomes popular. For example this can be used to signal MBone traffic.

A potential future application is layered video, where enhancement layers can be dropped in congestion. As the base layer is retained, performance is not badly affected. But there are concerns to such a use. McCanne et al. [MJV96] argue that, as drop priority does not badly affect end-to-end performance in congestion, there is no incentive for users to send less (in absence of pricing/policing). Additional traffic is dropped, thus wasting resources upto the bottleneck. They propose a solution for multicast group communication (can be simply extended to unicast case). Source sends different layers of a video on different multicast groups. Receivers independently join and leave these groups, based on perceived congestion. In times of congestion, users can leave groups corresponding to enhancement data. Such back off strategy leads to efficient utilization of network resources.

While this is true, the proposed feedback mechanism is an area of research. Such feedback may not react fast enough in case of large group. A drop priority can be provided that only filters *transient* congestion. We believe this can be accomplished using RED though we are not aware of any simulation results that prove this.

Priority can be used as congestion indication bits i.e., instead of dropping packets router may mark them and end-hosts use the feedback. Another use, suggested by Dave Clark, is to mark packets as 'in-profile' or 'out-of-profile' with respect to a traffic policing function, that was contractually negotiated between network and client. Routers may choose to exclude them from the profile policing computation.

4.3 Overview of mechanism

The aim of our implementation is to define a priority with wide applicability. In absence of a single method acceptable to all, we define a general mechanism that can be tuned to suit local requirements. We assign meaning to two additional bits as shown in Figure 8.

```
+---+---+---+---+
| L | P | R | I |
+---+---+---+---+

L : Link sharing class
    0 = link sharing class 0
    1 = link sharing class 1
P : Priority within link sharing class
    0 = low priority
    1 = high priority
R : Reserved for future use
I : Interactive class
    0 = non interactive
    1 = interactive
```

Figure 8: Priority Definition

The most significant two bits are considered only if I = 0. At the top level there are three classes - link sharing class 0, link sharing class 1, and interactive class. Each class is provided a relative apportionment of link bandwidth e.g., class 0 may have 60 %, class 1 has 10 % and interactive class has 30 %. The link sharing is implemented using weighted fair queueing algorithm [DKS89, CSZ92]. This provides isolation between classes, one class cannot interfere with traffic in another class. The exact allocation can be set by local administration. When some classes are not using their share, excess bandwidth will be used by other classes; but in times of congestion, no class can exceed its allocation. Classes can be assigned a weight of 0 %.

The maximum queue length for each top level class is configurable. Interactive class is served by using a proportionately smaller maximum queue length, compared to class 0 and class 1. Additionally it may be assigned a lower percentage of link bandwidth to control misuse. If D is desired maximum queueing delay at a node, and R is the rate allocation, maximum queue length L is given by $L = D * R$. If Interactive class is unable to use the allocated bandwidth due to higher packet loss, its assignment can be increased until the target throughput is achieved.

Within each of class 0 and 1, one of several methods can be selected by administration.

FCFS Queueing This simply implies absence of priorities within a class. It can be used if, only a link sharing mechanism is required.

Strict Ordering High priority is always queued ahead of low priority. In congestion, packets at the tail of the queue are dropped.

It can guarantee minimum delay and high availability for defense applications, network management data or routing protocol traffic. It starves low priorities and may be undesirable in common user applications. In addition, applications may react unfavourably to arbitrary reordering within a flow.

Ordering without Starvation It tries to limit the number of high priority packets that cut ahead of low priority packets. This avoids starvation of low priorities. We implement an algorithm similar to one suggested in [PP88].

Absolute Drop Priority Priority is used only in times of congestion, to drop low priority packets before any high priority packets.

It is similar to a cell loss probability bit in ATM. It can be used in applications like layered video or to indicate less important frames in a MPEG2 stream.

RED - Random Early Detection is a new congestion control technique for gateways proposed in [FJ93]. RED detects incipient congestion by computing an average queue length. When this exceeds a minimum threshold, an incoming packet is dropped (marked) with a certain probability, that is a function of the average queue length. Above an upper threshold, all arriving packets are dropped. RED replaces the drop-tail strategy used in routers. It offers various benefits - maintains low average queue size while allowing occasional bursts; number of dropped packets roughly correspond to bandwidth used by a connection; avoids global synchronization of many connections simultaneously backing off during congestion.

Priorities can be provided by assigning different drop probabilities to different classes [Bak96]. As increasing number of Internet gateways will use RED or its variant, we believe this scheme may find wide usage.

Below we discuss likely usage of the link shared classes 0 and 1. These can correspond to congestion-controlled and non congestion-controlled traffic respectively. Class 0 traffic backs off in response to congestion e.g., TCP. As TCP constitutes around 80 % of wide area Internet traffic, it may be assigned a relatively higher proportion of bandwidth. Additionally newer applications such as source- or receiver-based rate-adaptive video [BTW94, MJV96], which react to congestion by reducing the rate, can use this class.

Within class 0, further separation using RED with differential drop probability can be utilized in various ways.

- There are two classes of users, with one being offered premium service. This may require more than one P bit, though.
- In layered video as mentioned previously. Low priority has higher drop probability, so end-to-end performance is not badly affected in transient congestion. But if congestion persists, the *instantaneous* drop probability of high priority also increases due to increased average queue length. This leads to drop in high priority traffic too. Thus there is incentive for users to implement back off in absence of pricing/policing. This would require careful tuning of RED

parameters, based on actual measurements in an environment. A single P bit may be sufficient for this.

- Congestion-controlled rate-adaptive applications may be assigned a higher drop probability than TCP, because their response to congestion can be slow (due to large group size). An alternative is to use the class 1 for such traffic, and use RED separately in both classes 0 and 1.

Class 1 can be used for non congestion-controlled traffic. Many applications, such as multicast video, do not back off in response to congestion. Mixing such traffic with traffic that backs off in congestion only harms the latter. For example Mbone traffic can use the class 1. Though the best approach is to implement back off mechanisms in applications (we are considering only best effort traffic here), this requires changes to current applications, as also such mechanisms are an area of research. Such applications can be isolated from congestion-controlled class and can be provided a relatively smaller allocation.

An alternative way to use the mechanism, is using all top level classes for subscriber based link-sharing (thus semantics of I bit change). Our mechanism would allow three subscriber classes. Though it may be desirable to have more than 3 classes in this case.

Another use is to experiment with new service models as earlier mentioned. One service model where the class 1 can be useful is the 'predictive' or controlled load real-time service [CSZ92, Wro97]. Receivers dynamically adapt to increased delay upto a threshold. Network attempts to deliver end-to-end behaviour similar to that received by best-effort traffic, under *unloaded* condition. This is achieved using admission control. [CSZ92] shows that such service can be provided with FIFO queueing. This could be assigned a class 1 priority. The two classes based on P bit, can be provided different maximum delay bounds. The way this may operate is, senders use flow labels to identify individual flows. Ingress routers provide policing function. When a packet complies with the flow's negotiated traffic specification, its priority is set to a suitable class 1 priority. Internal routers need not look-up state for individual flows, and may simply use the received priority.

4.4 Implementation

The mechanism is implemented at the queues of outgoing interfaces. It can also be provided at other processing points concerned with allocation of resources, such as packet buffers or link layer connections [Bak95], but do not concern with such usage.

The output routines of various network protocols - IPv4, IPv6, IPX etc. invoke a common routine at the device layer for each outgoing packet. Device layer provides a generic interface to upper layers, and hides device specific features. It invokes a device specific output routine, if device idle, otherwise packets are queued in device queues. Queued packets are transmitted from an interrupt service routine, invoked when the device completes finishes transmission of current packet. Detailed description of device layer handling in Linux is provided in [Cox96].

Design Goals

The primary objective is to provide a mechanism to support priority based traffic classes. The implementation should have minimal per-packet processing overhead. Local administration should be able to dynamically configure the employed mechanism and related parameters.

Specifying Priority in Outgoing Packets

We provide a way for applications to specify priority on outgoing packets. Priority can be specified by setting the `sin6_flowinfo` member of following structure. It comprises the least significant 4-bits, of the most significant byte of sin6_flowinfo, considered in network byte order.

```
struct sockaddr_in6 {
    u_int16m_t sin6_family;      /* AF_INET6 */
    u_int16m_t sin6_port;        /* transport layer port # */
    u_int32m_t sin6_flowinfo;    /* IPv6 flow information */
    struct in6_addr sin6_addr;   /* IPv6 address */
};
```

This structure is passed to output functions - connect(), sendto() and sendmsg(). For transmitted packets of a passively accepted TCP connection, this can be specified during bind(). Additionally a socket option SO_PRIORITY is available to change the priority on a socket. A router uses the priority on received packets while forwarding on outgoing link.

4.4.1 Weighted Fair Queueing

The top level classes require apportionment of link bandwidth. Various algorithms have been proposed in literature to accomplish this [DKS89, Zha90, FJ95]. Demers et. al. [DKS89] propose a fair queueing (FQ) algorithm to allocate bandwidth fairly among number of traffic sources. This algorithm attempts to emulate a bit-by-bit round robin (BR) discipline which is always fair. They further generalize it to allow arbitrary bandwidth assignments. This modified algorithm is referred as weighted fair queueing in [CSZ92]. [Zha90] describe a clock based algorithm that emulates TDM discipline, while providing arbitrary assignments and statistical multiplexing. It is functionally similar to WFQ [ZK91]. [FJ95] propose a hierarchical link sharing mechanism based on use of estimator, general scheduler and link sharing scheduler.

It differs from WFQ in that the assignments are guaranteed over an interval of time, while WFQ attempts instantaneous guarantees.

We have implemented WFQ. A conceptual explanation is given here. Consider a BR discipline. Fair sharing implies sending one bit from each active source in one round. Weighted BR implies sending more than 1 bit from each source in a round, the exact number depending on individual allocation. Let

n_α : number of bits allocated to source α per each round
$R(t)$: total number of rounds upto time t
μ : linespeed of outgoing line
$A(t)$: set of active sources
$N(t)$: total number of bits sent by *active* sources per each round = $\sum_\alpha n_\alpha$, $\alpha \in A(t)$

Then $\partial R/\partial t = \mu/N(t)$. A packet of size P^α whose first bit is serviced at time t_0 will have the last bit serviced at time t such that $R(t) = R(t_0) + P^\alpha/n_\alpha$. Let packet i from source α and size $P_i{}^\alpha$ arrive at the queue at time $t_i{}^\alpha$, then the finishing round ($F_i{}^\alpha$) for this packet is given by

```
if ($\alpha \in A(t)$)
    $F_i{}^\alpha = F_{i-1}{}^\alpha + P_i{}^\alpha/n_\alpha$,
else
    $F_i{}^\alpha = R(t_i{}^\alpha) + P_i{}^\alpha/n_\alpha$
```

With more than one packets in the queues, ordering of the $F_i{}^\alpha$ values gives the order in which various packets finish transmission in a weighted BR scheme. The packet based WFQ algorithm is simply that whenever a packet is to be scheduled for transmission, one with the smallest value of $F_i{}^\alpha$ should be selected. This algorithm asymptotically approaches the fairness of the weighted BR scheme and the maximum instantaneous discrepancy is bounded by $P_{max}/\min(n_\alpha)$, where P_{max} is the maximum packet size.

There are two issues in implementation of the above algorithm -

- Computing the exact bit round in progress ($R(t)$) on packet arrivals is difficult. One solution proposed in [DH90] is to approximate the current bit round in progress, by the bit round in which current packet transmission finishes. They argue that the short-term unfairness introduced by this approximation would be similar in magnitude to that introduced by departure of packet based WFQ from weighted BR discipline. This considerably simplifies the implementation [Kes91].
- Within queues for class 0 and class 1, packets can be arbitrarily reordered, hence the finishing round number of a packet cannot be computed upon packet arrival, instead this is done when selecting a packet for transmission from the queues.

Organization of Queues

Before describing implementation of WFQ, we consider organization of the various queues. In our implementation, each interface is represented by a `struct device` that stores interface specific parameters such as name of interface, port, irq, mtu, etc., and pointers to device specific routines such as initialization, output etc. Each link shared class is represented by a `struct link_class` shown below. The `struct device` contains an array `class[3]` of type `struct link_class` whose elements correspond respectively to the three classes - class 0, class 1, and interactive. Each packet on an interface lies on the buffer queue of one of these classes and the packet at the head of each class's buffer queue is the one to be considered when selecting the next outgoing packet.

```
struct link_class {
    struct buff *prev, *next; /* head,tail pointers to buffer queue */
```

```
        unsigned char q_weight;     /* Relative bandwidth allocation */
        unsigned long q_len;        /* Length of queue in # of bytes */
        unsigned long nq_len;       /* Length of queue in # of pkts */
        unsigned long maxq_len;     /* Maximum length of queue in # of bytes */
        unsigned long fin_round;    /* Finish round number i.e. $F^\alpha$ */
        struct red_data red;        /* RED specific information */
    };
```

In addition to above, a packet on class 0 or class 1 queue also lies on another queue corresponding to the value of its P bit. There are four such queues represented by an array `pclass[4]` of type `struct pri_class` in `struct device`. `struct pri_class` contains head and tail pointers of corresponding (low/high) priority queue. Operations on this queue are described later.

WFQ Implementation

Each buffer contains a quantity `num_round` not part of the actual transmitted data, that corresponds to P_i^α / n_α above. The `fin_round` member of each link_class is set to zero during initialization. Two routines that implement WFQ are `dev_queue()`, invoked when a packet arrives on a device, and `dev_transmit()`, invoked when a packet is to be selected for link layer transmission.

```
/* This prevents wrap around */
# define MAXPRI 4000000000U

dev_queue(buff, dev) {
    i = link class queue for buffer, j = P bit of buffer;
    k = i + j;
    enqueue buff on dev->class[i];
    if (i != 2)
        enqueue buff on dev->pclass[k];

    if (dev->class[i].q_weight != 0)
        buff->num_round = buff->len/dev->class[i].q_weight;
    else
        if (dev->class[i].q_len != 0)
            buff->num_round = MAXPRI;
        else
            buff->num_round = MAXPRI - dev->class[i].fin_round;
}

dev_transmit(dev)
{
    unsigned long min_round = MAXPRI, tmp_round;
    int out_q = -1, empty_q = -1, empty_class[3];

    for (i = 0; i < 3; i++)
        if (dev->class[i].q_len != 0) {
            tmp_round = dev->class[i].fin_round +
                        dev->class[i].next->num_round;
            if (tmp_round <= min_round) {
                min_round = tmp_round;
                out_q = i;
            }
        }
        else
            empty_class[++empty_q] = i;

    if (out_q == -1)
        return; /* all queues empty */
    dev->class[out_q].fin_round = min_round;
    while (empty_q != -1)
        dev->class[empty_class[empty_q--]].fin_round = min_round;

    if (min_round >= MAXPRI)
        for (i = 0; i < 3; i++)
            dev->class[i].fin_round = 0;

    dev->ll_transmit(dequeue(dev->class[out_q].next));
}
```

4.4.2 Handling of P-bit

For non-interactive classes, the P-bit provides another level of priority handling. *Insertion order* of a packet on its `class[i]` queue depends on the currently configured queueing algorithm as described below. In each case, the packet is also queued at the tail of corresponding `pclass[k]` queue.

In FCFS queueing, each packet is queued at the tail of the corresponding `class[i]` queue. If maximum queue length is exceeded, a packet at the tail of the `class[i]` queue is discarded.

In Strict Ordering, a high priority packet is queued ahead of any low priority packet on the corresponding `class[i]` queue, while a low priority packet is queued at the tail of the `class[i]` queue. If maximum queue length is exceeded, packets are discarded from the tail of `class[i]` queue.

In Ordering without Starvation, each packet is associated with a *priority number*, locally within the node. The initial values in each of the two classes (separated by

P-bit) are configured by administration e.g., high priority assigns 4, low priority assigns 2. Packets are inserted on the `class[i]` queue sorted by their priority number. When a high priority packet cuts ahead of other packets, priority number of packets behind in the queue is incremented by 1. When maximum queue length is exceeded, packets are discarded from the tail of `class[i]` queue.

In Absolute Drop Priority, insertion in `class[i]` queue is similar to that in FCFS queueing. But when maximum queue length is exceeded, packets are discarded from the tail of low priority `pclass[k]` queue. If there are no low priority packets, then a packet at the tail of the high priority `pclass[k]` queue is discarded. Note that Strict Ordering also implies an Absolute Drop Priority.

In RED, the insertion order on `class[i]` queue is similar to that in FCFS queueing. But before insertion, the RED algorithm is invoked to determine if the packet should be dropped due to large average queue length. The exact RED algorithm is discussed in the next subsection.

4.4.3 Random Early Detection

Floyd and Jacobson [FJ93] propose the random early detection algorithm and discusses various aspects such as selection of parameters, simulation results, suggestions for efficient implementation, etc. A conceptual description of the algorithm follows.

RED uses a low pass filter based on an exponential weighted moving average of the queue size, to determine current level of congestion. This allows for occasional bursts in the traffic, as transient congestion does not significantly increase the calculated average. This is given by

$$avg \leftarrow (1 - w_q)avg + w_q q$$

The weight w_q is the time constant of the filter. If w_q is too large, the calculated average will be more sensitive to short-term congestion. If w_q is too small, the reaction to incipient congestion will be slow.

When the average queue size is less than a lower threshold (min_{th}), the packet is not dropped. If it exceeds an upper threshold (max_{th}), the packet is surely dropped. When the average is greater than the min_{th}, but less than max_{th}, the probability of dropping arrived packet increases with increase in the average queue size. The lower threshold should be set sufficiently large to maintain a high link utilization in the face of bursty sources. Value of the upper threshold affects the maximum average delay allowed at the gateway. The actual packet dropping probability (p_a) is calculated in two steps

$$p_b \leftarrow max_p(avg - min_{th})/(max_{th} - min_{th})$$
$$p_a \leftarrow p_b/(1 - count * p_b)$$

p_b varies linearly from 0 to max_p as the average varies from min_{th} to max_{th}. Here *count* is the number of packets since the last dropped packet. The computation for p_a guarantees that the inter-drop time (number of packets between two dropped packets) is represented by a *uniform* random variable. When the average queue size is halfway between min_{th} and max_{th}, the gateway drops, on the average, roughly one out of $1/max_p$ arriving packets [FJ93].

RED Implementation

Suggestions for efficient implementation of RED are given in [FJ93]. Gaynor [Gay96] describes an implementation of RED alongwith a proposed variation termed as derivative random drop. We measure the average queue length in terms of number of packets for simplicity, though a byte based RED can estimate the delay accurately.

The initialization routine is shown below.

```
#define TIMES        10
```

```
#define RANTBL_SZ    256
#define IWEIGHT      9    /* 1/512 */

int rantbl[RANTBL_SZ];
int *ranptr;

/* Initialization of RED parameters */
red_init(dev)
{
    struct red_data *red;
    ranptr = rantbl;
    red->scale_c1 = TIMES + log of (red->max_th - red->min_th) base 2;

    for (i = 0; i < 2; i++) {
        red = &dev->link_class[i].red;
        red->avg = 0;
        red->qtime = current time;
        red->count = -1;

        red->c1[0] = (1 << TIMES)/red->maxp_inv[0];
        red->c1[1] = (1 << TIMES)/red->maxp_inv[1];

        tmp = (red->min_th << TIMES)/(red->max_th - red->min_th);

        red->c2[0] = tmp/red->maxp_inv[0];
        red->c2[1] = tmp/red->maxp_inv[1];
    }
```

The parameter red->qtime is set to the current time whenever the queue length becomes zero during selection of a packet for link layer transmission. The following is invoked upon packet arrivals to determine whether the packet should be discarded or queued.

```
#define ZERO_TABLE_SZ    1200
int zero_que[ZERO_TABLE_SZ];

/* returns 1 to indicate drop */
random_early_drop(dev, i, p_bit)
{
    struct red_data *red = &dev->link_class[i].red;

    if (red->nq_len != 0)
        red->avg = red->avg +
                    ((red->nq_len << TIMES) - red->avg) >> IWEIGHT;
    else {
        diff = current time - red->qtime;
        if (diff < ZERO_TABLE_SZ)
            red->avg >>= zero_que[diff];
        else
            red->avg = 0;
    }

    if (red->avg < red->min_th)
        red->count = -1, return 0;
    if (red->avg >= red->max_th)
        red->count = -1, return 1;

    red->count++;
    pb = ((red->avg * red->c1[p_bit]) >> red->scale_c1) - red->c2[p_bit];

    if (red->count == 0) {
        ranptr = (ranptr + 1) % RANTBL_SZ;
```

or queued.

```
#define ZERO_TABLE_SZ    1200
int zero_que[ZERO_TABLE_SZ];

/* returns 1 to indicate drop */
random_early_drop(dev, i, p_bit)
{
    struct red_data *red = &dev->link_class[i].red;

    if (red->nq_len != 0)
        red->avg = red->avg +
                    ((red->nq_len << TIMES) - red->avg) >> IWEIGHT;
    else {
        diff = current time - red->qtime;
        if (diff < ZERO_TABLE_SZ)
            red->avg >>= zero_que[diff];
        else
            red->avg = 0;
    }

    if (red->avg < red->min_th)
        red->count = -1, return 0;
    if (red->avg >= red->max_th)
        red->count = -1, return 1;

    red->count++;
    pb = ((red->avg * red->c1[p_bit]) >> red->scale_c1) - red->c2[p_bit];

    if (red->count == 0) {
        ranptr = (ranptr + 1) % RANTBL_SZ;
        return 0;
    }

    if ((red->count * pb) >= *ranptr)
        red->count = 0, return 1;

    return 0;
}
```

4.5 Testing and Results

Testing of above mechanisms was conducted by running a number of experiments between two machines - a 100 MHz Pentium (dheeraj), and a 50 MHz PC-486 (godavari) both attached to the same ethernet LAN. In all these tests (except RED), a number of UDP sources on dheeraj send packets of size 1K, as fast as they can. The throughput obtained by each UDP source is measured at the *sending* end. This equals the number of packets each source could send during a fixed time interval divided by length of the interval. Note that the throughput calculated thus, correctly represents actual throughput only if no packets were not dropped due to queue overflows. The kernel was modified to maintain a count of total packets dropped for each priority class.

Various tests and their results are described below. For each UDP source the socket send buffer size is 64k.

a) Overhead of WFQ

Processing overhead for maintaining three link shared classes is compared to FIFO scheme with single queue and no priorities. A typical run consists of 3 UDP sources sending packets with priorities - 0, 8 and 1, over a 20 sec interval. Runs for the two schemes were intermixed in alternate order.

FIFO
Max queue length - 190 (hence no drop)
Average aggregate throughput over 5 runs - 1078 Kbytes/sec

WFQ
Max queue length - 80 : 80 : 80 (for each link sharing class)
Average aggregate throughput over 5 runs - 1072 Kbytes/sec

b) Link Sharing

Implementation of WFQ is tested. Typical run consists of 3 UDP sources sending packets at different priorities - 0, 8 and 1, over a 40 sec interval. The assigned weights for the three classes were 3 : 1 : 2.

Average throughput over 5 runs - 530 : 179 : 354 (Kbytes/sec)
Ratio - 2.96 : 1.0 : 2.0

c) Interactive Class

Configuration consists of two UDP sources at priorities - 0 and 1. Additionally two ping programs were run at the same time with priorities 0 and 1, to measure the round-trip time.

Total ping requests sent - 300 300

Maximum queue length - 40 : 20
Average round trip delay - 51.9 : 42.5 (ms)

d) Strict Ordering

Test setup consists of two UDP sources at priority 0 and 4 (class 0) sending packets over an interval of 20 sec.

Throughput obtained - 9 : 987 (Kbytes/sec)

e) Ordering without Starvation

All tests were over an interval of 20 sec with two UDP sources at priorities 0 and 4.

Init. Priority #	Throughput (Kbytes/sec)
0 : 12	385 : 442
0 : 24	346 : 600
0 : 64	188 : 742

f) Absolute Drop Priority

The maximum queue length was fixed to 60k. Hence for two UDP sources with priority 0 and 4 and send socket buffer size of 64k, the queue will overflow. The time interval of tests was 20 sec.

Total packets discarded due to overflow - 54577 : 0
Throughput - 185 : 866 (Kbytes/sec)

Maximum queue length	- 40 : 20
Average round trip delay	- 51.9 : 42.5 (ms)

d) Strict Ordering

Test setup consists of two UDP sources at priority 0 and 4 (class 0) sending packets over an interval of 20 sec.

Throughput obtained - 9 : 987 (Kbytes/sec)

e) Ordering without Starvation

All tests were over an interval of 20 sec with two UDP sources at priorities 0 and 4.

Init. Priority #	Throughput (Kbytes/sec)
0 : 12	385 : 442
0 : 24	346 : 600
0 : 64	188 : 742

f) Absolute Drop Priority

The maximum queue length was fixed to 60k. Hence for two UDP sources with priority 0 and 4 and send socket buffer size of 64k, the queue will overflow. The time interval of tests was 20 sec.

Total packets discarded due to overflow	- 54577 : 0
Throughput	- 185 : 866 (Kbytes/sec)

g) RED

A set of 12 TCP connections were setup between `dheeraj` and a 133 MHz Pentium (`skaold`) - one half with priority 0, and other half with priority 4.

w_q	- 1/512
min_{th}	- 2
max_{th}	- 10
max_p	- 1/16 : 1/64
Packets discarded by RED	- 174 : 2
Aggregate Throughput	- 465 : 492 (Kbytes/sec)

Chapter 5

Naming of Link-Local Addresses

5.1 Introduction

IPv6 link-local addresses are defined for use on a single link. Packets with a link-local source or destination address are not forwarded on other links. Routers configure a link-local address for each of their interfaces [HD97]. Additionally, standards [TN96] require that interfaces on hosts should configure a link-local address. A link-local address is formed by prepending the prefix FE80::0/64 to a 64-bit interface ID as shown in Figure 9 [HD97]. A link-local address has an infinite lifetime and does not time out.

```
|   10     |
|  bits    |         54 bits         |         64 bits            |
+----------+-------------------------+----------------------------+
|1111111010|            0            |       interface ID         |
+----------+-------------------------+----------------------------+
```

Figure 9: Forming a Link-Local Address

Link-local addresses can be used in various scenarios. In absence of routers, on-link nodes can communicate solely with the use of link-local addresses, thus enabling plug-and-play capability. Routing protocols for IPv6 use link-local addresses for communication between neighbouring routers [CFM97], as these remain unchanged in the face of site renumbering events such as change in service provider. For a similar reason neighbour discovery makes use of link-local addresses of routers in Router Advertisement and Redirect messages [NNS96].

But a facility for associating corresponding system names is lacking. It is inconvenient for humans to remember and work with hexadecimal representation of addresses as specified in [HD97], for example when specifying a server machine to contact to, or in the output from system utilities such as netstat, route etc. Hence a mechanism to associate names with addresses is desired. Various conventional methods for storing this mapping are - using a hierarchical set of distributed DNS servers [Moc87a, Moc87b]; static configuration in a system file (typically `/etc/hosts` on Unix systems); or static configuration on central server with dynamic distribution in restricted environment e.g., Network Information Service (NIS).

Storing system names for link-local addresses in the DNS requires a name server to determine relative locations of client and node for which lookup (or reverse lookup) is queried. Extending DNS to provide this information, would significantly increase its complexity. Manual mechanisms, either through a static file or configuration on a central server, are error prone and difficult to maintain in case of large number of on-link nodes or when the set of members is dynamic such as on wireless networks.

Harrington [Har97] proposes a solution to this problem. It defines a service on top of UDP that uses a multicast group address of link-local scope, to which all nodes join. The association at an individual node is advertised, periodically and/or in response to a query. Our implementation is based on this proposal. We modify the original proposal in several ways. The exact interaction between resolver and a linkname daemon is specified. Various options available to implementations are described. Based on our experience, recommendations for future implementations are provided.

In related work, Simpson [Sim95] defines a proposal to use ICMP messages for obtaining the Fully Qualified Domain Name (FQDN) [Moc87a, Moc87b] associated with IP (v4) addresses. More recently Crawford [Cra97] proposes a protocol similar to that proposed by Simpson [Sim95] for IPv6 networks. It is based on querying an IPv6 node to supply its FQDN, where the authorization based on DNS style of delegation is replaced by the routing infrastructure. It also uses ICMP messages. Though such a service shares few elements in common with the mechanism described in this chapter, both are conceptually different. We have implemented the mechanism on a Linux IPv6 stack running a modified version of the BIND 4.9.4 resolver.

The rest of the chapter is organized as below. Section 5.2 discusses the proposed solution and related issues. Section 5.3 provides a detailed description of the implementation.

5.2 Proposed Solution

We describe an outline of the proposed solution. Frame formats for client requests and server advertisements are specified. Various related issues such as the use of FQDN, multi-homed nodes and security are described.

Our goal is to provide a simple way to associate names with link-local addresses. A client that requires name to link-local address lookup (or vice versa), sends a query on a multicast group address of link-local scope (or the specific link-local address). A server on the node for which lookup was requested, replies to the request by sending a unicast reply directly to the client. Additionally, each server periodically advertises the local mapping at a node on the multicast group address. This information can be cached in recipient nodes for later use. This protocol works on top of UDP. A multicast group address, FF02::1:1, and UDP port, 1903 are reserved for use of this protocol [Har97].

Client Request

Figure 10 shows the format of a client request.

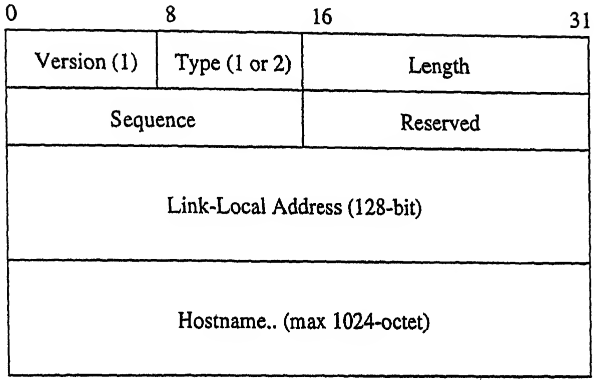

Figure 10: Format of Client Request Packet

Fields:

Version Version of the protocol, currently 1

Type Type of request

1 : Lookup a name for given address i.e., a reverse lookup

2 : Lookup an address for given name i.e., a forward lookup

Request of Type 1 is sent on link-local address being queried, while request of Type 2 is sent on the multicast group address.

Length Length of the packet in octets

Sequence A value used in matching requests to responses and to avoid duplicates

Link-Local Address Link-Local address for which hostname is requested

Hostname - Hostname for which link-local address is requested

Hostname is expressed as a FQDN. The reason for this is explained later. The maximum length of a FQDN is 1024 octets. For a request of Type 1, the Hostname should be empty, while for a request of Type 2, link-local address should be set to the unspecified address. In an alternative format, Hostname immediately follows the Reserved field, in a request of Type 2, and the link-local address is absent. But for simplicity, the field is retained and is set to the unspecified address.

Harrington [Har97] uses two fields for specifying the type of packet - a ADV/RQST field specifies whether it is an advertisement or request, and a TYPE field in a request specifies one of Type 1 or Type 2 request. Instead we use a single Type field for all three functions where a value of 3 indicates an advertisement.

Server Advertisement

The format of an advertisement is shown in Figure 11.

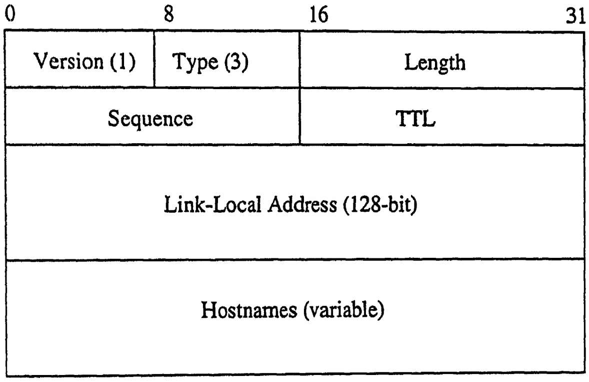

Figure 11: Format of Server Advertisement Packet

Fields:

Version Version of the protocol, currently 1

Type In an advertisement this contains a value of 3

Length Length of packet in octets

Sequence When responding to a request, this is copied from Sequence field of request, otherwise it is set to zero

TTL Time-to-live for advertised association, a value of FFFF specifies Permanent association which need not be timed-out, value of 0 indicates stale entry that should be flushed, and a value between these two specifies maximum number of seconds for which association may be considered valid

Link-Local Address The address for which association is advertised

Hostnames List of host names of the node corresponding to the link-local address; more than one names indicate aliases

Harrington [Har97] allows the specification of only a single name in each advertisement. We extend it to allow multiple names. Hostnames in the list are separated by delimiters, i.e., a null character, which does not occur in any name.

The solicited advertisement is sent on a client's unicast address, while periodic advertisements are sent to the multicast group. A mechanism to explicitly enable/disable either or both advertisements should be provided. Unsolicited advertisements are not strictly periodic, but a randomization is introduced between subsequent transmissions, to reduce probability of synchronization with advertisements from other on-link nodes [FJ94]. Each node maintains a timer. When an unsolicited advertisement is sent, the timer is reset to a uniformly-distributed random value between the configured `MinLLAdvInterval` and `MaxLLAdvInterval`; expiration of the timer causes next advertisement to be sent and a new random value being chosen.

Specifying Host Names

Host names in request and advertisement packets are specified as a FQDN for example *'godavari.cse.iitk.ernet.in'*. An alternative would be to use simple host names instead, for example *'godavari'*. But multiple nodes with the same simple hostname may co-exist on a link, such as *'www.cse.iitk.ernet.in'* and *'www.ee.iitk.ernet.in'*. A query on the name *'www'* leads to ambiguity as both nodes reply. Hence we require the use of FQDN in both request and advertisements. Hence in this example a query on just *'www'* will not be answered by any node.

Note that a resolver allows configuration of domain names and search lists in system files. These can be automatically appended to user specified names so that the final query, sent on the protocol is a FQDN, even though user specified a simple name.

Consider the case of a node not configured to use DNS. It is identified by a simple name unqualified by any higher level domains. Additionally resolver on such a node will not (usually) recognize search lists or domain names. When such nodes coexist on a link with nodes that use DNS, *users* on such nodes are responsible to specify the FQDN in forward lookups.

Harrington [Har97] suggests that requests be made general as possible and answers should be as specific as possible. For example a user may query on *'www'* and all nodes with simple name matching *'www'* respond, irrespective of their domains. Each response contains FQDN of the host. When multiple responses are received, some criteria can be used to select one of the responses, for example the longest response, the largest TTL, the first received response etc. However we *do not* recommend this, instead it is required that requests and responses both should be as specific as possible. This avoids ambiguity and simplifies implementations.

Multihomed Nodes

Various scenarios of multi-homed server and clients are specified in [Har97]. First a multihomed server should restrict advertisement of an address to the interface to which it is assigned. Second a client may have a neighbour with multiple interfaces on the same link or which has multiple link-local addresses assigned to the same interface. In this case the server may respond with several advertisements, one for each link-local address. A client should allow this and may use a criteria such as returning the largest TTL, or the first received address. In our implementation we return all the addresses for a name (or list of names) provided they identify the same node. The BSD socket API allows returning multiple addresses and aliases of a node in the same system call.

Consider the case of a multihomed client. Library functions for name/address lookups do not provide a way to specify the outgoing/incoming interface. Additionally a user's knowledge about the multiple interfaces may be limited. Hence a query is sent on *all* outgoing interfaces, and multiple responses are handled by returning the first received response. A different criteria can be used in this case. In absence of a mechanism to learn the incoming interface, the result may not always be desirable. Recommendation for a solution to this problem is made in Section 5.3.

Security

The above protocol is vulnerable to security threats, where a node masquerades as another node and advertises a malicious association. Note that as link-local addresses are configured automatically in most cases, the probability of this occurring through accident is very small. We consider solutions to the security problem here.

The IP Hop Limit field is set to 255 on both requests and advertisements. A receiving node checks this value before further processing. As each router decrements Hop Limit before forwarding, if the Hop Limit does not equal 255 at a receiver, the packet is discarded and a system warning logged. This protects against attacks from off-link nodes.

Usually a single link may have a level of trust, but in case of large number of on-link nodes, security can be a concern. In an environment where a node can overhear other conversations such as ethernet, other forms of attacks are possible and the threat is inherent to the type of link technology employed rather than this protocol.

For forward lookups , the search order may be set so as to query DNS first; if that fails a local file be consulted (`/etc/hosts`); and invoke this protocol after other methods have been attempted and failed. For reverse lookups, a local file can be searched first, if that fails, then the protocol may be invoked. Additionally, clients may completely ignore multicast advertisements, and use only direct queries and responses. If other nodes cannot overhear unicast traffic (such as on ATM), only the intended node receives a query, and sends a unicast reply. This minimizes the amount of threat.

Additionally a client may monitor responses. When multiple responses are received, it can increase the log level. The most effective solution to the security problem is to use the IPv6 security mechanisms [Atk95c], Authentication Header [Atk95a] and/or Encapsulating Security Payload [Atk95b]. But a standardized method to provide security associations required by these mechanisms is not available.

5.3 Implementation

We describe the implementation of naming support for link-local addresses in detail. We used the Linux 2.1.21 operating system as target operating system. The resolver was a modified version of BIND 4.9.4 available in [Met].

Design Goals

The primary goal is to provide a simple mechanism for associating system names with link-local addresses. The new functionality should be transparent to applications and they should work unmodified. There should be few changes to the resolver code. It should be possible to configure the mechanism to suit link characteristics and local requirements.

We consider how these goals are achieved and how they affect our implementation.

- Applications invoke routines in the standard C library (`libc`) for commonly used functions. Many operating systems, including Linux, support dynamically linked libraries. This allows modifying library functions without recompiling programs, if the function call interface is not modified. As the resolver is part of `libc`, modifying the resolver results in a new version of this library and application transparency is obtained.

- There should be few changes to resolver code. This was suggested in [Bou96]. It recommends using a cache file similar in format to `/etc/hosts` and of a dynamic nature, which resolvers can search when lookup is desired.

  We refer to this file as `/etc/llcache`. When a matching record is not found in `/etc/llcache`, a request is sent to a linkname daemon *on the same node*. The daemon forwards requests to the multicast group or specific link-local address, and receives advertisements. It modifies `/etc/llcache` on receipt of a valid advertisement. The resolver searches the file again after a delay. If a matching record is found, it returns success, else an error is returned.

  This results in minimal change to resolver and reuse of routines for parsing `/etc/llcache`. We revisit this issue towards end of this section. This implementation can be further simplified by completely eliminating direct queries. Resolver searches the cache once, and if no entry is found, it returns error. Servers work only in periodic advertisement mode. But this would require selection of suitable timers for the periodic advertisements. If the value is too large, the mechanism responds slowly to changes. If it is too small, it results in network congestion and wastage of resources.

- Configurability is provided in the implementation. The two types of advertisements - unsolicited and query responses can be individually enabled/disabled. Various timers can be specified when the daemon is started. Additionally resolver provides some amount of configurability.

5.3.1 Background

In this subsection we discuss the API provided by a resolver, resolver configuration and format of the cache file.

API Functions

Gilligan et. al. [GTBS97] discusses the library functions for forward and reverse lookups. A new function is specified for IPv6 hostname to address lookups.

```
#include <sys/socket.h>
#include <netdb.h>

struct hostent *gethostbyname2(const char *name, int af);
```

af specifies the address family and is set to AF_INET6 for IPv6. If successful, the function returns addresses in a struct hostent, otherwise it returns a NULL pointer. struct hostent is defined in <netdb.h>.

```
struct hostent {
    char *h_name;         /* official name of host */
    char **h_aliases;     /* alias list */
    int h_addrtype;       /* host address type */
    int h_length;         /* length of address */
    char **h_addr_list;   /* list of addresses */
};
#define h_addr h_addr_list[0]   /* address, for backward compatibility */
```

The conventional function for forward lookup in IPv4 is

```
    struct hostent *gethostbyname(const char *name);
```

This function returns IPv6 addresses instead of IPv4 addresses, if following resolver initialization is invoked by applications

```
    #include <resolv.h>

    res_init();
    _res.options |= RES_USE_INET6;
```

Internally gethostbyname() actually invokes gethostbyname2() for the IPv6 lookup. A protocol independent function, getaddrinfo(), is described in [GTBS97]. In our implementation this also invokes gethostbyname2() for IPv6 forward lookups.

Reverse lookup is provided by following function

```
    #include <sys/socket.h>
    #include <netdb.h>

    struct hostent *gethostbyaddr(const char *src, int len, int af);
```

A protocol independent function is also defined, getnameinfo(), which invokes the above function in our implementation. Thus changes to the resolver are restricted to two within functions gethostbyname2() and gethostbyaddr().

Resolver Configuration

Resolvers provide a range of configuration options. Important ones relevant to this discussion are summarized. A search order can be specified in a system file. Name of this file differs on different operating systems, example names are /etc/svc.conf, /etc/nsswitch.conf, /etc/host.conf. This specifies the order for lookups such as DNS/Local/Linkname, DNS/Linkname/Local etc. The BIND 4.9.4 used in our implementation *does not* provide such an option and the order is fixed in implementation. We implement the search order DNS/Local/Linkname.

The file /etc/resolv.conf allows specification of a domain or list of domains. This is used to automatically qualify a user specified simple name or incomplete FQDN, before actual lookup.

As the underlying protocol (UDP) is unreliable, the resolver retransmits a query some number of times if no response is received. The number of retries and delay between retries is specified by two variables _res.retry and _res.retrans. By default these are set to 4 and 5 seconds respectively.

Format of cache file

The cache file is similar in format to /etc/hosts. In case of IPv4, this consists of one record per line. Each record contains a numeric address in dotted quad notation, followed by hostname and aliases on the same line. A '#' indicates that rest of a line is a comment. For IPv6, this format is extended to allow multiple addresses per host. Each record contains a list of addresses, followed by a list of host names.

An example cache file is shown below

```
# This is the cache file for ipv6 link local addresses. It is
# dynamically modified by the linkname daemon. Do not modify
# this manually

fe80::c00f:8f fe80::ffff:fffe godavari.cse.iitk.ernet.in godavari  # 0 2
fe80::df48:9d dheeraj.cse.iitk.ernet.in dheeraj  # 863451875 2
```

This file is modified by the daemon. New records are appended towards the end. To delete a record, the first character on a line is converted to '#'. Some information is shown within comments after each record. This is ignored by resolver, but used by daemon. The first number indicates maximum time for which record is valid, while the second number identifies the corresponding link. When the number of deleted records exceeds a maximum count, the file is selectively purged.

5.3.2 Resolver Implementation

The linkname mechanism is invoked in two ways - from gethostbyname2() after a forward lookup using DNS and local file fails; and in gethostbyaddr() when a reverse lookup using local file fails. A simplified algorithm is shown in Figure 12.

```
parse_cache();
if (found)
    fill and return struct hostent;

for (i = 0; i < _res.retry; i++) {
    send_query();
    sleep(_res.retrans);
    parse_cache();
    if (found)
        fill and return struct hostent;
}
return NULL;
```

Figure 12: Resolver Algorithm for Linkname mechanism

In a forward lookup, the above algorithm is executed successively for each name, formed using search lists, when the previous name fails. The cache file is locked before parsing and unlocked after parsing completes. This prevents daemon from modifying the file while resolver parses the file. send_query() sends a query to the local daemon. The destination address of this query is the loopback address, ::1, and UDP port is set to 1903.

5.3.3 Daemon Implementation

Initialization

Initialization performed during startup is described below. A linkname daemon obtains the list of multicast capable interfaces and corresponding link-local address(es) (if any). FQDN of the local node and its aliases are determined. The cache file is created and initialized with information about the local mapping i.e., link-local addresses and names of the local node.

A UDP socket is opened, bound to the wildcard address and port 1903. Though it binds to the wildcard address, all packets with a destination address not equal to the loopback address or the multicast address FF02::1:1 are silently discarded. On each multicast capable link, it joins the multicast group FF02:1:1, using a `setsockopt()`. If periodic advertisements are enabled, an `alarm()` timeout is set for the initial timer.

Processing of Requests and Sending Advertisements

A request can be received from either a local resolver or a remote daemon. Local requests are verified as follows

- Request must be of type 1 or 2
- In a reverse lookup (Type 1), length of Hostname must be 0; in a forward lookup (Type 2), link-local address must be unspecified
- In a Type 1 request, the address must be link-local
- In a Type 2 request, length of Hostname must be non-zero
- Minimum Length of packet should be 24 octets

- Length of Hostname should not exceed maximum length of a FQDN (1024 octets)

If any of these conditions is false, the packet is silently discarded. Following are the steps in further processing of *local* requests. following

- Set version to 1
- Increment a global counter and set Sequence number to its value. Keep track of the sequence number in a bitmap, so a Sequence number of S, corresponds to the bit (S *mod* MAX_BITMAP_SIZE) in the bitmap
- Send on all outgoing (multicast capable) links with a hop limit of 255. Hop limit can be specified as ancillary data to `sendmsg()` [ST97]. The destination address for a Type 1 request is specified as the requested link-local address, while for a Type 2 request it is specified as FF02::1:1

Verification checks on a *remote* request include those mentioned for a local request, additionally the version must be 1 and received Hop Limit must be 255. Further processing occurs as follows.

- For a forward query, if the Hostname matches a local name or one of the aliases, an advertisement is sent separately for each of the link-local address assigned to the incoming interface
- For a reverse lookup, if the Link-local address in packet matches any of the link-local address assigned to the incoming interface, an advertisement is sent for the particular address
- All the local names and aliases are provided in each advertisement
- Sequence number on advertisement is copied from Sequence number on the query

- The response is sent directly to a requesting node, with hop-limit set to 255 and outgoing interface same as incoming interface of request

In addition to above, unsolicited advertisements are periodically sent on all interfaces on the multicast destination address. Each advertisement contains a link-local address assigned to the interface and list of host names and aliases.

Processing of Received Advertisements

Following are verification checks on receiving an advertisement.

- Version must be 1
- Source address must not be the loopback address
- If Sequence number is non-zero, the advertisement must not a duplicate retransmission
- Hop limit must be 255
- Advertised address must be link-local
- Any of the advertised names must not exceed 1024 octets

Following are the steps in further processing of an advertisement.

- Lock the cache file
- Search cache file for an entry with the same interface, as the incoming interface of advertisement, and the same list of hostnames as provided in the Hostnames field of the advertisement

- If no matching entry found, append a new entry in the cache file if advertised TTL is non-zero

- If a matching entry was found, it is deleted and a new entry is formed by merging with received information

- In the previous step, if advertised TTL is 0, a new entry is not appended

- Unlock the cache file

- If the minimum timeout changed as a result of above processing, set the system `alarm()` to a new timeout

5.3.4 Recommendations

Based on the above implementation experience, we offer several recommendations. The packet format can be further simplified by eliminating the Length field. Length can be obtained from the UDP length information returned by calls such as `recvfrom`

Locking of Cache File

Our implementation of file locking is rudimentary. It can be improved by implementing 'readers/writer' mutual exclusion with priority to writer

We reconsider the recommendation to minimize change in resolver code. The solution as we have implemented has a problem - it requires locking of cache file While this operation is done in the resolver or the daemon, there is no mechanism to prevent misbehaving users from locking the file and cause starvation.

A better solution would be *not* to use a cache file. Instead this information can be maintained in-core by the daemon, and provided upon query from resolver. We believe that while this increases resolver complexity, the amount of change is *not* significant. Additionally it speeds up lookup as file operations are eliminated.

Use of a '.link' pseudo-domain

Harrington [Har97] proposes the use of a new pseudo-domain, *'.link'*. User may suffix a *'.link'* to the name in forward lookups, such as *'godavari.cse.iitk.ernet.in.link'*. A resolver can interpret this to mean that linkname protocol should be invoked *first*, before other methods. But such use of a new pseudo-domain is discouraged [Lis97a]. Specification of a search order can be provided through defining suitable resolver options, instead of overloading DNS like names.

While we agree with this, there exists another scenario where such a facility can be beneficial. As mentioned earlier, the present API does not provide a mechanism to specify outgoing/incoming interface in name/address lookups. An extended form of the *'.link'* method can be used to provide this without modifying the API. We describe our proposal below.

Instead of a *'.link'* domain, suffixes of the form *'.link<id>'* should be allowed, where *id* indicates the index of a particular interface. The BSD API for IPv6 provides a uniform way to identify interfaces at a node using an interface index [GTBS97]. This can be used to specify the outgoing/incoming interface. We consider the two cases - a forward query and a received advertisement.

- In a forward query, if no such domain is suffixed or the interface index is 0 (an index 0 is not valid for any interface), the normal processing as described previously occurs i.e., query is sent on all outgoing interfaces. But if a non zero and existing index is specified, query must be sent only on the particular interface. The Hostname in the packet should not carry the pseudo-domain though.

- A received advertisement includes names and aliases of the advertising node. The API provides the ability to receive multiple names of a node. In addition to the received list of names, another list of names can be constructed by the daemon by suffixing each received name with the *'.link<id>'* domain, where *id* is set to that of the incoming interface.

Chapter 6

Conclusions and Future Work

We looked at salient features of IPv6 which are targetted to achieve a wide set of objectives laid out for a next generation IP. A study of three experimental issues was carried out in this thesis. Below we summarize what has been achieved and areas of further work.

Host Anycast Support

A solution is discussed for the problem with host anycast addresses that they cannot be used as source address of datagrams and as destination address in ongoing stateful communication. Requirements at both end-points of a communication, the anycast host and another host communicating with the anycast host, are specified. We focus on mechanisms in the two transport protocols, TCP and UDP. API extensions are suggested to provide flexible usage of host anycasting by applications. This mechanism allows many TCP/UDP applications to work unchanged. Applications that maintain state across multiple TCP connections or on top of UDP, or which use the end-point addresses at the application layer will need to be modified. The solution is implemented in and tested in a LAN environment.

More work needs to be done for secure communication using anycast address. Interaction with AH needs to be provided. Research is required to design a solution for key distribution in an anycast group. A recent standard [EK97] provides storage of authenticated public keys in DNS. It may be used to learn a public key of an anycast group during an initial query on anycast group name. Currently no mechanism is available to determine the required action when a datagram is received for which corresponding security association does not exist. It may be possible to query DNS to obtain keys in this case but this introduces complexity in protocol processing. Research is also required to solve the problem with Internet wide routing of host anycast addresses.

Priority Support

We discuss the definition of priorities in IPv6. Priority is loosely defined to permit flexible use. Issues related to a priority mechanism and likely usage are discussed. We utilize two additional bits and implement various algorithms. Top level classes are isolated using WFQ and priorities within these classes are separated using various queueing algorithms - FCFS queueing, Strict Ordering, Ordering without Starvation, Absolute Drop Priority and RED. These mechanisms are implemented in and tested in a LAN environment between two machines. Various parameters can be configured dynamically, thus permitting tuning of the mechanism to suit individual requirements.

In further work, the various mechanisms should be tested in realistic conditions of network configuration and application behaviour to determine the actual benefit of a priority mechanism. Work needs to be done for providing an efficient policing mechanism. Our implementation ignores interaction with other network protocols such as IP, IPX, etc., on the same node. A hierarchical link sharing mechanism should be implemented to completely isolate different network protocols. This can be provided using WFQ recursively or with class based queueing [FJ95]. Additionally some problems have been cited with the RED algorithm proposed in [FJ93]. This needs to be looked into.

Naming Link-Local Addresses

The problem of naming IPv6 link-local addresses is discussed. The name to address association cannot be stored in DNS. A proposed solution to this problem is discussed. This proposal is modified in several ways to provide simplicity and flexibility. Exact interactions between resolver and server are specified. Finally recommendations are provided to further improve the protocol operation. These mechanisms have been implemented on a Linux IPv6 stack running a modified version of the BIND 4.9.4 resolver. It was tested between two hosts.

The implementation can be further improved by implementing the suggested recommendations. Also, interaction with IPv6 security mechanisms can be provided. Harrington [Har97] suggests an extension where a node may act as proxy for a set of nodes and provides a proxy bit for this. This mechanism can be incorporated with few modifications in our implementation.

Appendix A

Modifications Required

Following is a list of bugs in the present implementation.

- Special handling is provided when a packet with anycast destination address is received on a socket in TIME_WAIT state. There is a useful BSD violation of the host requirements RFC - if a new request matches a socket in TIME_WAIT state and the initial send sequence number is greater than any sequence number used on the matched socket, then the connection is moved to CLOSED state and the segment is demultiplexed again to find another matching socket. Presently we provide such action in the case when a request is received with a anycast destination address. Instead this should be done only if the socket in TIME_WAIT was the result of a connection establishment on the anycast address.
- The special handling for UDP based on use of `state_req` flag needs to be implemented.
- In absence of AH, the `IPV6_RCVANY` flag for TCP and UDP can be assigned only one of two values, true or false.

- The API implementation lacks two functionalities - `ioctl()` to determine if given address is anycast and a `setsockopt()` to return peer's anycast address in `recvmsg()` and `getsockopt()`.

- There is a single set of RED variables. It is only implemented in the class 0 priority. It should be moved to the per device structure, and should have two sets of variables one each for class 0 and class 1 priorities.

- The check for measuring maximum queue length is currently packet based, it should be byte based. On overflow only a single packet is discarded, this should consider more than one packets.

- Request for a reverse query is sent on the multicast group address, instead of specific link-local address. Advertisements are not restricted to a specific interface.

- The maximum size of a domain is set to 255 instead of 1024 octets, and each hostname is assumed to be null terminated.

- The randomization in periodic advertisements needs to be provided.

- Handling of SIGHUP signal needs to be provided, to initialize interface information without having to stop the daemon.

- Configuration options cannot be set on the command line, instead daemon program has to be recompiled. Solicited responses cannot be disabled presently.

- A correct way to determine name and aliases of local host is required as DNS does not return this information.

References

[Alm92] P. Almquist. Type of Service in the Internet Protocol Suite. RFC: 1349, July 1992.

[Atk95a] R. Atkinson. IP Authentication Header. RFC: 1826, August 1995.

[Atk95b] R. Atkinson. IP Encapsulating Security Payload (ESP). RFC: 1827, August 1995.

[Atk95c] R. Atkinson. Security Architecture for the Internet Protocol. RFC: 1825, August 1995.

[Bak95] F. Baker. Requirements for IP Version 4 Routers. RFC: 1812, June 1995.

[Bak96] F. Baker. Mail to IPng Mailing List regarding use of IPv6 Flow Label for tag switching, October 11, 1996. Archive available by subscribing to 'ipng' at majordomo@sunroof.eng.sun.com.

[Bel89] S. Bellovin. Security Problems in the TCP/IP Protocol Suite. *ACM Computer Communications Review*, 19(2), March 1989.

[Bel96] S. Bellovin. Defending against sequence number attacks. RFC: 1948, May 1996.

[BM93] S. Bradner and A. Mankin. IP: Next Generation (IPng) White Paper Solicitation. RFC: 1550, December 1993.

[BM95] S. Bradner and A. Mankin. The Recommendation for the IP Next Generation Protocol. RFC: 1752, January 1995.

[Bou96] J. Bound. Mail to IPng Mailing List regarding draft-ietf-ipngwg-linkname-01.txt, February 13, 1996. Archive available by subscribing to 'ipng' at majordomo@sunroof.eng.sun.com.

[BR96] J. Bound and P. Roque. IPv6 Anycasting Service: Minimum Requirements for End Nodes. Internet Draft, June 1996. Work in Progress.

[Bra89] R. Braden. Requirements for Internet hosts – communication layers. RFC: 1122, October 1989.

[BTW94] J. Bolot, T. Turletti, and I. Wakeman. Scalable Feedback Control for Multicast Video Distribution in the Internet. *Proceedings of SIGCOMM'94*, 24(4), October 1994.

[CD95] A. Conta and S. Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC: 1885, December 1995.

[CESZ91] R. Cocchi, D. Estrin, S. Shenker, and L. Zhang. A Study of Priority Pricing in Multiple Service Class Networks. *SIGCOMM'91 Conference*, 21(4), September 1991.

[CFM97] R. Coltun, D. Ferguson, and J. Moy. OSPF for IPv6. Internet Draft, March 1997. Work in Progress.

[Cox96] A. Cox. Network Buffers and Memory Management. Linux Journal, October 1996.

[Cra97] M. Crawford. IPv6 Name Lookups Through ICMP. Internet Draft, March 1997. Work in Progress.

[CSZ92] D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. *SIGCOMM'92 Conference Proceedings*, 22(4), October 1992.

[DH90] J. Davin and A. Heybey. A Simulation Study of Fair Queueing and Policy Enforcement. *ACM Computer Communications Review*, 20(5), October 1990.

[DH95] S. Deering and R. Hinden. Internet Protocol Version 6 (IPv6) Specification. RFC: 1883, December 1995.

[DKS89] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *SIGCOMM'89 Symposium*, 19(4), September 1989.

[EK97] D. Eastlake and C. Kaufman. Domain Name System Security Extensions. RFC: 2065, January 1997.

[FJ93] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4), August 1993.

[FJ94] S. Floyd and V. Jacobson. The Synchronization of Periodic Routing Messages. IEEE/ACM Transactions on Networking, April 1994.

[FJ95] S. Floyd and V. Jacobson. Link Sharing and Resource Management Models for Packet Netoworks. *IEEE/ACM Transactions on Networking*, 3(4), August 1995.

[FLYV93] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. RFC: 1519, September 1993.

[Gay96] M. Gaynor. Proactive Packet Dropping Methods for TCP Gateways. unpublished manuscript, October 1996. Available at URL *http://www.eecs.harvard.edu/˜gaynor/final.ps.*

[GN96] R. Gilligan and E. Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. RFC: 1933, April 1996.

[Gro94] P. Gross. A Direction for IPng. RFC: 1719, December 1994.

[GTBS97] R. Gilligan, S. Thomson, J. Bound, and W. Stevens. Basic Socket Interface Extensions for IPv6. RFC: 2133, April 1997.

[Har97] Dan Harrington. Link Local Addressing and Name Resolution in IPv6. Internet Draft, January 1997. Work in Progress.

[HD97] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. Internet Draft, June 1997. Work in Progress.

[Hin97] Robert Hinden. Meeting Minutes of IPng working group at the Memphis IETF, April 10 & 11, 1997. URL *http://playground.Sun.COM/ipng/minutes/IPng-Meeting.April97.txt.*

[HOD97] R. Hinden, M. O'Dell, and S. Deering. An IPv6 Aggregatable Global Unicast Address Format. Internet Draft, May 1997. Work in Progress.

[Hui94] C. Huitema. The H Ratio for Address Assignment Efficiency. RFC: 1715, October 1994.

[IEE97] IEEE. Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority, March 1997. URL *http://standards.ieee.org/db/oui/tutorials/EUI64.html.*

[Jay96] M. Jayaram. Implementation of IPv6 for Linux. Master's thesis, Indian Institute of Technology, Kanpur, Computer Science & Engineering Department, June 1996.

[Joh96] M. Johnson. *The Linux Kernel Hackers' Guide.* On-line document, URL *http://www.redhat.com:8080/HyperNews/get/khg.html*, 1996.

[JP96] D. Johnson and C. Perkins. Mobility Support in IPv6. Internet Draft, November 1996. Work in Progress.

[Kes91] S. Keshav. On the Efficient Implementation of Fair Queueing. *Journal of Internetworking: Research and Experience*, 2(3), September 1991.

[KM87] C. Kent and J. Mogul. Fragmentation Considered Harmful. *SIGCOMM '87 Workshop on Frontiers in Computer Communications Technology*, 17(5), August 1987.

[Lis97a] IPng Mailing List. Discussion on draft-ietf-ipngwg-linkname-01.txt, January 31 to February 25, 1997. Archive available by subscribing to 'ipng' at majordomo@sunroof.eng.sun.com.

[Lis97b] IPng Mailing List. Discussion on IPv6 priority field usage and Qry on Traffic Policing (IPv6 header changes), March 4 to June 21, 1997. Archive available by subscribing to 'ipng' at majordomo@sunroof.eng.sun.com.

[MDM96] J. McCann, S. Deering, and J. Mogul. Path MTU Discovery for IP version 6. RFC: 1981, August 1996.

[Met] Craig Metz. inet6 applications kit for Linux. URL ftp://ftp.inner.net/inet6-apps-0.04.tar.gz.

[MJV96] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driver Layered Multicast. *Proceedings of SIGCOMM'96*, 26(4), August 1996.

[Moc87a] P. Mockapetris. Domain Names - Concepts And Facilities. RFC: 1034, November 1987.

[Moc87b] P. Mockapetris. Domain Names - Implementation And Specification. RFC: 1035, November 1987.

[NNS96] T. Narten, E. Nordmark, and W. Simpson. Neighbor Discovery for IP Version 6 (IPv6). RFC: 1970, August 1996.

[PK94] C. Partridge and F. Kastenholz. Technical Criteria for Choosing IP The Next Generation (IPng). RFC: 1726, December 1994.

[PMM93] C. Partridge, T. Mendez, and W. Milliken. Host anycasting service. RFC: 1546, November 1993.

[Pos80] J. Postel. User Datagram Protocol. RFC: 768, August 1980.

[Pos81a] J. Postel. Internet Protocol. RFC: 791, September 1981.

[Pos81b] J. Postel. Transmission Control Protocol. RFC: 793, September 1981.

[PP88] W. Prue and J. Postel. A Queueing Algorithm to Provide Type-of-Service for IP Links. RFC: 1046, February 1988.

[Sim95] W. Simpson. ICMP Domain Name Messages. RFC: 1788, April 1995.

[SPG97] S. Shenker, C. Partridge, and R. Guerin. Specification of Guaranteed Quality of Service. Internet Draft, February 1997. Work in Progress.

[ST97] R. Stevens and M. Thomas. Advanced Sockets API for IPv6. Internet Draft, March 1997. Work in Progress.

[Ste90] R. Stevens. *Unix Network Programming.* Prentice Hall, Englewood Cliffs, NJ, USA, 1990.

[Ste94] R. Stevens. *TCP/IP Illustrated: The Protocols*, volume 1. Addison-Wesley, Reading, MA, USA, 1994.

[TH95] S. Thomson and C. Huitema. DNS Extensions to support IP version 6. RFC: 1886, December 1995.

[TN96] S. Thomson and T. Narten. IPv6 Stateless Address Autoconfiguration. RFC: 1971, August 1996.

[WC92] Z. Wang and J. Crowcroft. A Two-Tier Addess Structure for the Internet: A Solution for the Address Space Exhaustion. RFC: 1335, June 1992.

[Wro96] J. Wroclawski. The Use of RSVP with IETF Integrated Services. Internet Draft, October 1996. Work in Progress.

[Wro97] J. Wroclawski. Specification of the Controlled-Load Network Element Service. Internet Draft, May 1997. Work in Progress.

[Zha90] L. Zhang. VirtualClock: A New Traffic Control Algorithm for Packet Switching Networks. *SIGCOMM'90 Symposium*, 20(4), September 1990.

[ZK91] H. Zhang and S. Keshav. Comparison of Rate-Based Service Disciplines. *SIGCOMM'91 Conference*, 21(4), September 1991.